THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Throughput and energy efficiency of lock-free data structures: Execution Models and Analyses

ARAS ATALAR

Division of Networks and Systems
Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Throughput and energy efficiency of lock-free data structures: Execution Models and Analyses
Aras Atalar

ISBN: 978-91-7597-783-6
Doktorsavhandlingar vid Chalmers tekniska högskola
Ny series nr 4464

ISSN: 0346-718X
Technical report 161D
Department of Computer Science and Engineering
Distributed Computing and Systems Group

Division of Networks and Systems
Chalmers University of Technology
SE-412 96 GOTHENBURG, Sweden
Phone: +46 (0)31-772 10 00

Author e-mail: aaras@chalmers.se

Printed by Chalmers Reproservice
Gothenburg, Sweden 2018

Throughput and energy efficiency of lock-free data structures: Execution Models and Analyses

Aras Atalar
Division of Networks and Systems, Chalmers University of Technology

ABSTRACT

Concurrent data structures are key program components to harness the available parallelism in multi-core processors. Lock-free algorithmic implementations of concurrent data structures offer high scalability and possess desirable properties such as immunity to deadlocks, convoying and priority inversion. In this thesis, we develop analytical tools to model and analyze the throughput and energy consumption of concurrent lock-free data structures. We start our study with a general class of lock-free data structures. Then, we target more specialized designs for lock-free queues. Finally, we focus on the search data structures that possess different characteristics compared to previously mentioned data structures.

Performance of lock-free data structures: This thesis contributes to the problem of making ends meet between theoretical bounds and actual measured throughput. As the first step, we consider a general class of lock-free data structures and propose three analytical frameworks with different flavors. Analyses of this class also cover efficient implementations of a set of fundamental data structures that suffer from inherent sequential bottlenecks. We model the executions and examine the impact of contention on the throughput of these algorithms. Our analyses lead to optimization methods on memory management and back-off strategies.

Performance and energy efficiency of lock-free queues: We take a step further to model the throughput of lock-free operations and their interaction. Considering shared queues, as a key paradigm for data sharing, operations (*Enqueue*, *Dequeue*) access the opposite ends of a queue. Same type of operations might contend with each other on a non-empty queue. However, all types of operations are subject to interaction when the queue is empty. We first decorre-

late the throughput of dequeuers' and enqueuers' into several uncorrelated basic throughputs, and reconstruct the main throughputs as a function of these basic throughputs. Besides, we model the power dissipation and integrate it with the throughput estimations to extract the energy consumption of applications that utilize lock-free queues.

Performance of lock-free search data structures: Lock-free designs that utilize fine-grained synchronization have produced efficient implementations of search data structures. These designs reveal different characteristics compared to the previous set of lock-free data structures with inherent sequential bottlenecks. We introduce a new way of modeling and analyzing the throughput of search data structures under stationary and memoryless access patterns.

Keywords: Concurrency, Lock-free, Data Structures, Parallel Computing, Performance, Throughput, Energy Efficiency, Modeling, Analysis

List of publications

This thesis is based on the work contained in the following publications and their extended versions. Result I, II, IV (that are presented in this thesis) are extended versions of the first, second, and fourth articles that are listed below, respectively. Result III is the work presented in the third article that is detailed below.

- **Aras Atalar**, Paul Renaud-Goud and Philippas Tsigas, "Analyzing the Performance of Lock-Free Data Structures: A Conflict-Based Model", *In the Proceedings of* 29^{th} *International Symposium on Distributed Computing (DISC 2015)*, pages 341-355, Springer 2015.

- **Aras Atalar**, Paul Renaud-Goud and Philippas Tsigas, "How Lock-free Data Structures Perform in Dynamic Environments: Models and Analyses", *In the Proceedings of the* 20^{th} *International Conference on Principles of Distributed Systems (OPODIS 2016*), pages 1-17, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik 2016.

- **Aras Atalar**, Anders Gidenstam, Paul Renaud-Goud and Philippas Tsigas, "Modeling Energy Consumption of Lock-Free Queue Implementations", *In the Proceedings of* 29^{th} *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2015)*, pages 229-238, IEEE Press 2015.

- **Aras Atalar**, Paul Renaud-Goud and Philippas Tsigas, "Lock-Free Search Data Structures: Throughput Modeling with Poisson Processes", *Under Submission*.

The following results have also been achieved during my PhD studies and they are not part of the thesis:

- Christoph W. Kessler, Lu Li, **Aras Atalar** and Alin Dobre, "XPDL: Extensible Platform Description Language to Support Energy Modeling and Optimization", *In the Proceedings of* 44^{th} *International Conference on Parallel Processing Workshops (ICPPW 2015*), pages 51-60, IEEE Computer Society 2015.

- Adones Rukondo, **Aras Atalar** and Philippas Tsigas, "2D-Stack: A scalable lock-free stack design that continuously relaxes semantics for better performance", *In the Proceedings of* 37^{th} *ACM Symposium on Principles of Distributed Computing (PODC, Brief Announcement)*, pages 407-409, ACM 2018.

Acknowledgments

First of all, I would like to thank my supervisor Philippas Tsigas for his encouragement and guidance. This thesis would not have been possible without his support.

I thank the members of my graduate committee: Peter Demaschke, Agneta Nilsson, and Marina Papatriantafilou, for their kind support and help.

I am honored to have Guy Blelloch as my opponent and sincere thanks to the members of my grading committee: Dan Alistarh, Bengt Jonsson, Alessia Milani, Ioannis Sourdis.

I want to thank my former mentors and advisors: Etkin Abi, Cevdet Aykanat, Gokhan Erbay, Michael Gerndt, Michael Rauh, Isaias Urena and Paul Renaud-Goud.

I am grateful to the Swedish Research Council for funding me and to EXCESS Project and its members from whom I have gained a lot.

I thank my friends that I had the chance to work with: Ivan, Anders, Fazeleh, Lazaros, Adones, Lu, Ali, and Paul. Thanks to my officemates: Vincenzo, Nasser, Hannah and special thanks to my roommate Thomas.

I would like to thank former and current members of the DCS group and department for the excellent work environment: Babis, Eva, Marianne, Rebecca, Peter, Rolf, Magnus, Elad, Olaf, Tomas, Bapi, Valentin, Bashr, Thomas Jr., Iosif, Ioannis, Zhang, Nhan, Giorgos, Alijoscha, Bei, Elena, Boel, Evangelis, Stavros, Madhavan, Prajith, Stefano, Albin, Amir, Petros and Chloi and to all youngsters that have joined recently and to all experienced members.

Last but not least, I would like to thank my parents and my family for their

support, unconditional love and more.

Aras Atalar
Göteborg, September 2018

Contents

List of Figures

Part I

INTRODUCTION

1 Introduction

Moore's Law [1] suggested an exponential growth in the number of transistors per unit area on the integrated circuits. Doubling the transistor density every year allowed vendors to implement more complicated designs that can switch state in shorter time periods. For almost 40 years, increasing the clock frequency of a processor was the way to improve the performance. However, it was unlikely to maintain this exponential growth in the longer term.

In the last two decades, we observed a shift from single-core chips to multi-core chips as it became infeasible to increase the clock frequency and complexity of a single core. The main triggers of this shift were the need to: (i) reduce power consumption and eliminate cooling-related issues; (ii) tolerate the high off-chip memory latency with concurrent accesses; (iii) decrease the circuit complexity of cores; (iv) expel some physical limitations regarding the

transistor size and the wire lengths.

Hardware vendors have changed their objective from increasing the core performance to increasing the chip performance [2]. They employ more and more cores on a single chip where individual cores might need to collaborate to exploit available computing power.

Naturally, these developments in hardware pose new challenges for applications utilizing the hardware. Applications have to be developed to harness the available computing power in an effective way which brought out the multi-core programming challenge. Today, multi-core processors are exploited by a broad spectrum of devices ranging from supercomputers to a vast number of smartphones through almost all personal computers. Therefore, small improvements on the software might lead to significant outcomes.

Multi-core processors are composed of multiple computing units that we will refer to as processors (interchangeably cores). The execution entities that are running on processors will be referred to as processes (interchangeably threads) in the context of this thesis. Multiple processors can execute computations simultaneously which will be referred to as parallelism, and the overlapping execution of processes introduces concurrency, where the execution of processes can interleave and influence the behavior of each other [3]. None of these concepts implies another. For example, the execution of a process can be interleaved by the other concurrent processes that are scheduled on the same sequential processor or multiple processors can run independent tasks in parallel without the possibility of influencing each other. In this thesis, we study the performance of programs that are executed on multi-core shared memory system, in the settings with both parallelism and concurrency.

In the multi-core parallel programming model, multiple processes collaborate in order to complete tasks, each executing a program asynchronously [4] (*i.e.* the execution can be halted and the execution of instructions can be delayed). The processes communicate, if needed, by accessing shared objects in the shared memory. In the course of a computation, processes access shared objects in a sequence of atomic events in which some subsequence of the events might need to occur or look like they occur atomically. However, events from

multiple processes can interleave each other in concurrent executions. If not synchronized, some of these interleavings might lead to inconsistencies. Consider the case with two processes that want to increment a shared counter. A process reads the counter and writes back the incremented value. It can be observed that some interleavings of these four events would lead to an inconsistent state in which the counter seems to be incremented only once. To avoid this, some consistent order of the events on the shared objects can be enforced by synchronization.

Synchronization techniques define the way of accessing shared objects by adhering to some correctness (safety) and progress (liveness) guarantees. In simple terms, the correctness guarantee states that something bad does not happen and the progress guarantee states that something good keeps happening [4].

In the sequential setting, operations on the object are defined by the pre- and post-conditions, that also constitute the intended behavior, namely the sequential specification (correctness) of the object. Unsurprisingly, many correctness properties in the concurrent setting, where actions on the shared object from different actors can be interleaved, are based on the equivalence to some sequential behavior of the object, *e.g. Linearizability* [5], *Sequential Consistency* [6]. In simplified terms, linearizability requires that there exists a legal sequential execution (a legal total order with respect to sequential specification) that preserves the irreflexive partial order of the real concurrent execution. For each process, an invocation on the object is followed by the response. Let $inv.op$ and $res.op$ denote the invocation and response times of an operation op in the concurrent execution, where every invocation has a corresponding response. The irreflexive partial order $\prec$ on the set of operations O that are executed by any process is given by: $\forall(a, b) \in O^2, a \prec b \Leftrightarrow inv.b > res.a$. Linearizability is widely adopted as a correctness guarantee for the concurrent data structures [4]. However, the correctness guarantees are only meaningful if they come together with some progress guarantees as it is trivial to be correct without performing any action.

The progress guarantees can be split into two categories; *blocking* and *non-blocking* that lead to blocking and non-blocking synchronization mechanism,

respectively. The distinction can be stated simply as follows: in blocking synchronization, a process can block another hence the delay of a process can delay others, whereas in non-blocking synchronization, the delay of a process cannot delay others [4]. We will provide technical information about the taxonomy and possible implementations of non-blocking and blocking synchronization mechanisms in the following sections.

We can use an analogy with a conversation, where a shared conversation object evolves with the contribution of multiple agents. Due to the interleaving possibility, people indeed employ synchronization mechanisms generally to keep the shared object in a correct state (no misunderstandings) while allowing progress (some understanding should keep happening). The conventional way for this is to adopt a blocking approach. People talk one after the other, waiting for the talking person to finish and then ask for the token to continue. However, this approach might have some limitations. For example, a less important topic could be discussed while a more important one has to wait because it is blocked by the less important one. Alternatively, the phone of the talking person might ring, and others need to wait for the phone call to finish even though the original conversation has nothing to do with the phone call. On the other hand, optimistic people apply non-blocking approaches where people start and keep talking because they believe that their ideas will be understood eventually. In these cases, it could be quite tricky to satisfy the correctness and progress conditions, and other techniques have to be applied such as repetition. Although it is clear that non-blocking approaches eliminate the limitations of their blocking counterparts, their efficiency is poorly understood. The efficiency of talking styles could be evaluated according to different metrics, *e.g.* throughput (the number of expressed ideas in a unit of time) and total energy consumed during the discussion.

This thesis considers the throughput and energy consumption of the lock-free (non-blocking) synchronization mechanisms in shared memory multi-core systems, by focusing on the concurrent data structures. The thesis is organized into three parts: Introduction, Results and Conclusion. We provide background information in the first part. Section 1.1 examines the basics of the blocking

and non-blocking synchronization mechanisms, also discusses the atomic primitives that are needed to build such mechanisms. Section 1.2 briefly explains the relevant architectural features of modern multi-core processors. Section 1.3 discusses the lock-free data structures: design techniques, important performance metrics, performance models and analyses. Lastly, the contributions of this thesis are addressed in Section 1.4. Research articles are presented in the second part of the thesis and we conclude with the conclusion and future work.

1.1 Synchronization Techniques

1.1.1 Blocking Synchronization

A common way to implement synchronization is to use blocking approaches that are based on the notion of critical section. Critical sections are often implemented with the help of locks that mark the code blocks that need to possess the property: mutual exclusion. The mutual exclusion property ensures the correctness of the shared objects, by guaranteeing that the owner of the lock will not be disturbed by other lock-seekers. To implement a synchronization mechanism, it should be coupled at least with the deadlock-freedom property to satisfy some progress guarantees. The definitions of mutual exclusion and deadlock freedom property are given below:

- *Mutual Exclusion* [4]: at most one process can execute the concerned code block at a time.
- *Deadlock Freedom* [7]: if there exists a process that tries forever to enter its critical section execution, then there exists an infinite number of critical section executions.

Generally speaking, the blocking approaches are relatively more straightforward to implement. However, they have some inherent limitations mainly originating from the dependency of lock-seeker processes to the lock-owner. A crucial drawback is that blocking approaches cannot tolerate the failure of a single process. If a process fails while holding a lock, it might inhibit the system

progress by blocking the other processes that request to execute the corresponding critical section. Another undesired effect is priority inversion where a low priority task holds a lock that a high priority task needs. Think of the following situation. The high priority task yields its quanta to the low priority task so that it can finish its critical section and release the lock. Then, a medium priority task comes, preempts the low priority task and executes before the high priority task. Finally, the deadlock possibility should be considered when multiple locks are used. This could be painful especially during the integration of different components of large-scale software. Therefore, non-blocking approaches might be preferable.

1.1.2 Non-Blocking Synchronization

The synchronization mechanism regulates the accesses to the shared object and the term *non-blocking* reflects the nature of an attempt to access the shared object. It states that an attempt to access or modify the shared object cannot block or be blocked by another process, regardless of the state of the system. However, it does not specify the outcome of the attempt. Non-blocking mechanisms can be classified according to the progress guarantees that they provide, as follows:

- *Wait-Free Synchronization* [8]: A synchronization technique is wait-free if it ensures that every process will continue to make progress in the face of arbitrary delay (or even failure) of other processes.

- *Lock-Free Synchronization* [9]: A synchronization technique is lock-free if it ensures only that some process always makes progress.

- *Obstruction-free Synchronization* [10]: A synchronization technique is obstruction-free if it guarantees progress for any process that eventually executes in isolation.

The stronger the progress guarantee, the more complex the synchronization mechanism gets. Wait-free solutions are generally computationally costly.

Obstruction-free solutions are weak concerning the guarantees in a concurrent environment. Roughly speaking, lock-freedom is located around the sweet spot of this trade-off, which explains the attention it has received in the industrial applications, *e.g. Intel's Threading Building Blocks Framework* [11], *Java concurrency package* [12], *Microsoft .NET Framework* [13].

1.1.3 Atomic Primitives

Implementing a synchronization mechanism (in the asynchronous multi-core systems) based on only atomic reads and writes are either impractical (because of space and time complexity, *e.g. Bakery algorithm [14]*) or even impossible for many types of synchronization [4]. To overcome these limitations, multi-core architectures often provide atomic primitives that atomically conduct a set of operations, *i.e.* such that an intermediate state during the execution of the instruction cannot be observed. These primitives can be used to synchronize processes efficiently.

Atomic primitives can be classified according to their consensus number [8], which is as a measure of their power. The consensus number specifies the maximum number of processes for which the primitive (together with read/write registers) can solve the wait-free consensus problem, where each process must agree on an input value after a finite number of its own steps [15]. Wait-free consensus is critical because it allows processes, that are communicating asynchronously, to keep a consistent view of the shared objects. Atomic instructions, with a large enough consensus number, can be used to modify the shared object by having a wait-free consensus on each step of the progress. The semantics of some widely used atomic instructions are provided in Figure 1.1.

In [8], it has been shown that a primitive cannot be implemented by other primitives with lower consensus numbers. *Compare-And-Swap* has an infinite consensus number, and it can be used to construct wait-free implementation of any abstract data type. Although it is very convenient for implementing non-blocking synchronization mechanisms, it should still be used with care since its redundant usage can lead to performance degradations. Another limitation

```
Compare-And-Swap (var, old, new)
    if var = old then
        var ← new
        return true
    else
        return false
```

```
Test-And-Set (var)
    if var = 0 then
        var ← 1
        return true
    else
        return false
```

```
Fetch-And-Add (var, value)
    old ← var
    var ← var + value
    return old
```

Figure 1.1: Semantics of Atomic Primitives

is that it suffers from the *ABA* problem. A variable value was read to be *A* and was given as the old value parameter to the *Compare-And-Swap*. Then, the value is changed to *B* and then back to *A* again by concurrent operations. *Compare-And-Swap* cannot distinguish this configuration from a configuration without any modification by concurrent operations. Consequently, the successful *Compare-And-Swap* might lead to a state that is not intended by design. This limitation is often overcome through *memory management* schemes in the context of concurrent data structures.

1.2 Multi-core Architectures

Multi-core processors are composed of multiple independent computational processors (cores), each of them can execute a program asynchronously and communicate with the other cores through the shared memory. To obtain better performance, shared memory is often organized in a hierarchy where the main memory, residing at the top, is supported by a set of shared and/or private caches that are located closer to the cores for faster access. Modern multi-core proces-

sors show a great variety in the organization and properties of these components in terms of the size of memory/caches modules, memory hierarchy depth, access policies, number of cores, number of sockets, number of memory modules and the access latencies of these modules by different cores (could be uniform or non-uniform), *etc.*. Here, two critical issues need to be addressed from the synchronization perspective.

The first one is the *cache coherency protocol* (*e.g.* MESIF [16]) that assures the consistency of the copies of the data (in the unit of cache lines) in the multiple partially shared and private caches. In many cases, the protocol is represented with a state machine where each copy of the cache line is assigned to a state. The cache coherency protocol plays the lead role in the implementation, hence the latency of the atomic primitives. It enforces *exclusive ownership* of the target cache line by the core (through invalidation of the other copies if needed) during the execution of the atomic primitive. Therefore, the latency of the atomic primitives is determined based on the state of the target cache line copies and their locations in the memory hierarchy [17]. Assume *Core* 0 has a cache line in exclusive state in its private cache. Then, it would be cheaper for *Core* 0 to execute an atomic primitive on the cache line compared to *Core* 1 because *Core* 1 first needs to fetch and invalidate the copy in *Core* 0's private cache to bring the cache line to exclusive state in its private cache. The latency of this cache coherency communication between local caches of *Core* 0 and *Core* 1 depends on the location of the *Core* 0 and *Core* 1. For instance, this latency is higher for the cores that reside in different sockets compared to the ones that reside in the same socket in a NUMA architecture. It is possible to build cost models for many architectures which is necessary for our purposes because the cost of the memory events have a direct impact on the throughput and the energy consumption of lock-free data structures.

The second issue is the *memory consistency model* of the multi-core processor. Sequential consistency [6] is a natural option since the memory access ordering strictly follows the program order for each core. Nevertheless, for performance reasons, it is common to relax this model and allow reordering of some memory accesses, only if they are referencing to different locations in the

memory. *Total Store Order* (TSO) memory model allows loads to be moved ahead of stores. As explained in [18], this can be done by employing store buffers. For example, a store operation that experiences a huge stall time (*e.g.* due to the state of the cache line, the data needs to be fetched from a remote location), can be recorded to the store buffer and the execution can continue with the next read access that completes in a short time interval. Later on, when the cache line arrives, the data can be copied from the store buffer to the cache line. In this way, one can take advantage of the reordering to overlap the stall with other operations. This optimization does not create any problem for the sequential programs but might lead to undesirable behavior for the concurrent ones. We consider the program in Figure 1.2: if no reordering can take place, two zeros cannot be observed in the output. By contradiction, if this happens, then "print a" should occur before "a←1", meaning that "b←1" occurred before "print b", which is impossible. However, this situation might happen with the reordering of memory accesses: Process 0 waits for b, and in the meantime, prints a, and Process 1 prints b while waiting for a.

$a, b \leftarrow 0$

Process 0	**Process 1**
$b \leftarrow 1$	$a \leftarrow 1$
print a	print b

Figure 1.2: Memory access reordering

Memory fence instructions are provided by multi-core processors to ensure the completion of the pending memory accesses for a core before the initiation of the following accesses. While implementing synchronization mechanisms, these memory fences can be used to overcome the reordering issue together with the consideration of the memory consistency model. In return, memory fences could decrease significantly performance as they do eliminate optimization possibilities. But the behavior of the program becomes more evident with them, which facilitates the performance predictions.

1.3 Lock-free Data Structures

Data structures organize the data in a way to allow efficient access. Concurrent ones also allow multiple processes to share data and communicate asynchronously. For some applications, this type of communication can be useful. For example, it allows for overlapping the communication and computation phases of different processes which might reduce the communication overhead if the overhead is a function of the number of processes that are communicating simultaneously. Also, it can help parallel applications to dynamically distribute and balance the load by eliminating the need to wait for the slowest process (e.g. work stealing with deques [19], producer/consumer design pattern with concurrent queues [20]).

Concurrent data structure operations are designed by employing synchronization mechanisms which determines the associated correctness and progress properties. As mentioned before, an operation can be classified as blocking or non-blocking based on the progress guarantees that it provides.

Lock-based operations are blocking, and they rely on the mutual exclusion property for correctness. Critical sections mark the code blocks that need to be executed atomically and traditionally are protected by locks. There are two main approaches to implement lock-based concurrent data structures. As a coarse-grained approach, a process can lock the whole data structure to operate in isolation. However, this approach might block some concurrent operations unnecessarily. A coarse-grained lock can delay the progress of concurrent operations even if they operate on disjoint parts of the data structure or serialize some steps of the operations that do not need to be executed in isolation. Fine-grained approaches avoid this wastefulness by locking possibly the minimum amount of shared resources for the minimum amount of time. Hence, they lead to more efficient data structure designs. Although it is easier to implement and reason about lock-based data structures compared to non-blocking data structures, lock-based approaches suffer from the limitations that are mentioned in Section 1.1.

We have categorized the non-blocking methods in Section 1.1.2 based on

the level of established progress guarantees. Obstruction-free designs provide progress guarantees that are insufficient for a concurrent environment. On the other end of the spectrum, wait-free designs might exhibit insufficient performance metrics to satisfy strong progress guarantees. Lock-free data structures designs are generally efficient and scalable. Also, lock-free algorithms often exhibit wait-free progress guarantees in practice [21]. These facts highlight the convenience of lock-free data structures for practical usage.

1.3.1 Design Techniques

Lock-free implementations employ optimistic conflict control and guarantee system-wide progress. In contrast to pessimistic lock-based approaches, processes do not signal their presence before the operation, work independently and check at the end whether their independent work is invalidated. As a result, delays occur only if there is an actual conflict between concurrent processes. For example, an inactive preempted process cannot delay another process which is possible when one relies on mutual exclusion.

Three steps form the basic block to design a lock-free data structure operation: accessing the concurrent data structure to determine its state, preparing the desired changes to the concurrent data structure locally and trying to apply them to the shared state in an atomic way (thanks to an atomic primitive). When included in a retry loop, the basic block can be repeated until the desired changes are applied to the concurrent data structure.

Similar to the coarse- and fine-grained locking approaches, lock-free data structures are designed in many different ways. *Universal constructions* are design techniques that can transform any sequential object into a safe concurrent object. As a coarse-grained approach, one can always rely on the universal construction described by Herlihy in [9], where it is shown that any abstract data type can get a lock-free implementation based on a single retry loop that applies the whole operation with a single successful atomic primitive.

In simple terms, the construction is realized in three steps: a process (i) accesses (via a shared pointer) to the object; (ii) copies the object and applies

the sequential operation to the copied object; (iii) tries to apply the changes to the shared state by updating the shared pointer to the updated copy with an atomic primitive, and repeats the three steps until the third step is successful which happens only if the shared pointer is not updated by another process between step one and three. This approach introduces two problems for the large objects. It is inefficient to copy a large object, and the potential parallelism might be inhibited because the updates can conflict even if they modify the disjoint parts of the copied object (*i.e.* the implementation is not disjoint-access parallel [22]). Although this construction emphasizes mostly the computability aspect in asynchronous concurrent environments, it can be used as a basis to design efficient implementations of some fundamental abstract data types that have inherent sequential bottlenecks. This can be done by updating only a small portion (memory words that host the bottleneck) of the data structure while the old and new versions are sharing the untouched portion of the data structure.

A popular example is Treiber's lock-free stack [23]. Its operations (push and pop) are realized with a single retry loop, both following a very similar structure. Figure 1.3 provides the structure of the push operation of Treiber's stack. The stack is formed of a linked list of nodes where the top variable points to the first node. A push operation takes a new node as its parameter and appends it to the top of the stack. One can observe the three steps: (i) read the top pointer to determine the first element of the stack; (ii) prepare the new desired state locally by setting next field of the new node to the address of the first element; (iii) try to commit this state as the new state of the data structure with a *Compare-And-Swap* (CAS) on the top pointer to update it with the address of the new element. These steps are repeated in a retry loop until a successful CAS, whose failure would imply the existence of another successful concurrent operation.

For some other abstract data types, more practical designs apply the basic block in multiple, finer steps that gradually carry the data structure to the desired state. As in the fine-grained locking, this reduces the conflicts between different operations and provide better performance. However, it is harder to obtain the lock-free progress guarantee property when the operations are com-

Push ($newNode$)
  while (! $success$)
    $oldNode \leftarrow top$
    $newNode.next \leftarrow oldNode$
    $success \leftarrow CAS(top, oldNode, newNode)$

Figure 1.3: Treiber Stack Push Operation

pleted in multiple steps. The strategy here is to leave a sign to the other processes regarding the state of the operation after each step so that they can take action accordingly in order to guarantee the system-wide progress. Having encountered an incomplete operation a process might (i) ignore and start its own operation, if possible; (ii) try to help (often not a selfless type of help) the incomplete operation before executing its own operation; (iii) try to merge the incomplete operation with its own operation at hand.

For example, one can think of Delete operation on the lock-free skip list [24]. This operation might require updates on multiple pointers in order to entirely detach the deleted element from the skip list. All these updates are not applied atomically but gradually each leaving a sign regarding the state of the operation. First, the element is logically deleted with a mark. This mark leaves a sign to other processes so that they can determine the state of the incomplete operation in case they are operating in the vicinity of the deleted element. This knowledge allows them to avoid modifications that would lead to inconsistent states and take action (help for the next steps of the incomplete delete operation or ignore if possible) accordingly. In the same vein, the remaining steps of the operation are gradually executed until the element is completely detached from the skip list.

Loosely speaking, helping might create focal contention points, and ignoring might introduce additional work [25]. Some combination of these techniques is often used to design efficient lock-free data structure operations depending on the data structure type or the usage context. There are numerous lock-free implementations of various abstract data types with different design

choices: skip lists [24, 26], binary trees [27, 28], stacks [23, 29, 30], queues [20, 31–34], vectors [35], bags [36], deques [37, 38], priority queues [39, 40], hash tables [41, 42], linked lists [43, 44]. This variety complicates the gathering of lock-free data structures under a unified generic design.

1.3.2 Throughput

A common metric for measuring the performance of lock-free data structure is throughput, defined as the number of successful operations per unit of time. For generic lock-free algorithms, the execution time of a single operation cannot be bounded. It is then more natural to consider sequences of operations instead, since all the operations in the sequence will not encounter bad executions. In this context, the performance is often measured with the average system throughput over a sequence of operations.

We are interested in the throughput of concurrent lock-free data structures, and the underlying impacting factors that drives this throughput. These impacting factors are viable for the performance of *all* lock-free data structures that we consider in this thesis, but the significance of these impacting factors differs based on the characteristics of the data structure and on the context they are used in.

Retry loop and hardware conflicts: Lock-free operations cannot be blocked but some parts of an operation can be repeated due to the existence of conflicting concurrent operations within the retry loops. Under high contention, *retry loop conflicts* occur, and this retry loop conflicts might lead to a second type of conflict, that we refer to as *hardware conflicts*. Retry loops contain atomic primitives that can stall other memory accesses (atomic primitives, read/write that access the same memory word) while getting executed. When multiple atomic primitives are issued in the same time interval, they serialize (the latency of memory accesses expands due to stall time) and this leads to significant performance degradation.

Under high contention but in the absence of hardware conflicts, failing retry loop iterations introduce additional useless work to the failing (repeating) pro-

cess but they *often* do not decrease the system performance. This is because two successful retry loop iterations cannot overlap in time and the successful one cannot be obstructed by failing retry loop iterations if there are no hardware conflicts. Therefore, increasing the number of processes in the retry loop would merely increase the number of failed retry loop iterations, but would not harm the system performance. However, hardware conflicts do not only introduce useless work (through waiting time) to the failing process but also harm the system performance. Think of a sequence of serialized *Compare-And-Swap* instructions: while a process will operate a successful *Compare-And-Swap* (due to the progress guarantee), the rest of the processes in the retry loop are doomed to failure. If they are scheduled to execute their *Compare-And-Swap* when the possibly successful one is pending, the system performance is reduced. Failing *Compare-And-Swap*s do not change the content of the memory word but only obstruct the successful one. This impact can escalate with the increase in the number of processes in the retry loop. It gets harder to get out of the retry loop for a successful process (*i.e.* the ratio failing/successful *Compare-And-Swap* increases), and the additional delay of the successful operation leaves more space for new processes to arrive at the retry loop, that increases the contention further. This interplay might create *hot spots*.

In such cases, *back-off* strategies can be used to convert this harmful work (failing *Compare-And-Swap*) to a harmless but useless one. Failing processes can back-off, instead of retrying, to let the others succeed with less blockage. The back-off would increase the system performance, but its amount should be tuned since a small amount might be ineffective and large amount might lead to an underutilization of the resources.

Lock-free data structures that have inherent sequential bottlenecks are more prone to retry loop conflicts, thus to hardware conflicts. For such data structures, accesses are concentrated on a small number of memory words. For example, a plain stack is accessed via its top pointer by all of its operations—in the same way, queue operations access either the head or the tail of the queue. Regardless of the size of the stack (the number of elements inside), all operation accesses the top pointer. This characteristic might lead to contention in the form

of hot spots whose severity is determined by the number and access rate of the processes that are performing the operations.

Number of Loads/Stores and Cache Misses: Previously mentioned factors (*retry loop conflicts* and *hardware conflicts*) are specific to use cases with high concurrency. There are also performance impacting factors that are not related to concurrency and appear both in sequential and concurrent executions. For example, consider a binary tree (or a skip list, a hash table) that might lead to accesses on a large number of different memory words over a sequence of operations. Even in the absence of concurrency-related conflicts, one needs to estimate *the number of memory word accesses per operation* and connected to this, in the practical domain, the *cache capacity misses*. This estimation might not be trivial for some data structures like a binary tree, in contrast to simpler data structures such as stacks or queues.

On the bright side, this characteristic (accesses are *not* concentrated on a small number memory words) might turn out to be an advantage in the concurrent executions (*i.e.* leading to a good scalability) because the processes might spread to different shared memory words (for example to the different branches of a binary tree); this reduces the possibility of retry loop and hardware conflicts, and in turn, the possibility of hot spots. If we assume that the number of memory words is much bigger than the number of processes (excluding extremely imbalanced access patterns), the retry loops and hardware conflicts would have a *negligible* impact on the performance of such data structures.

This does not mean that these data structures are immune to contention since every modification still requires a consensus. This consensus leads, on the logical side, to a consistent view of the lock-free data structure that is accessed and modified by multiple processes concurrently in a non-blocking manner. On the practical side, achieving this consensus and spreading the information during and after its achievement impacts performance of all processes in the system. This impact is merely small compared to the other mentioned impacts.

The struggle of processes executing the same retry loop is often viewed as the major source of contention when they try to propose different values for the same consensus object within the same time frame (which leads to retry

loop conflicts and hardware conflicts). The impact of contention on the learners (the processes that read the modified memory word) is less apparent since the contenting events may not occur close in time. More clearly, consider two consecutive accesses to a memory word j by a process i that happen at time t_0 and t_1, respectively. For the access at t_1, process i would experience a *coherence cache miss* if memory word j is modified by another process in between t_0 and t_1. Search data structures, *e.g.* hash tables, skip lists, trees, contain multiple consensus objects (nodes), and this characteristic leverages the impact of the retry loop contention against the *coherence contention on the learners* dramatically.

Through this thesis, we address these performance impacting factors in various configurations. We focus on the *retry loop conflicts* (and their subsequent performance impactor *hardware conflicts*) for data structures that have sequential bottlenecks (*e.g.* stack, queue, priority queue, counter). We set parameters for our models to analyze the congestion points so as to cover a large set of possible lock-free data structure designs, contention levels, and use cases. For search data structures, we focus on the main impacting factors, the most significant of which are *the number of memory accesses*, *capacity* and *coherency cache misses*. We construct a model based on these impacting factors and show that it can be initiated with different abstract data types (*e.g.* skip list, hash table, binary tree).

1.3.3 Energy Consumption

Energy consumption has recently become a vital optimization criterion, for several reasons [45]. For example, the electricity cost for the operation and the cooling of data centers has reached to a significant amount. Fixed battery capacity is a constraint for mobile devices. Moreover, ecological footprints of data centers, supercomputers, personal and mobile devices necessitate a balance between low energy use and high performance.

Energy consumption is obtained by the time-integral of power consumption that is classically split into two parts; *static* and *dynamic*. The switching activity

of the transistors, based on the characteristics of the executed program, leads to the dynamic part of the power dissipation. On the other hand, the static part originates from the leakage effects that also exist when the transistors do not change state, hence it is independent of the executed program.

At the hardware level, modern multi-core processors employ several low power techniques to reduce the energy consumption. When the system is idle, the dynamic part of the power is negligible. *Power gating* technique can be used to minimize the static leakage part of the power. Computing units can be put to power saving states (sleep states) and can be *activated* back when needed. For an active system, computing units can be put into different operation states by changing the clock frequency. *Dynamic Voltage and Frequency Scaling* (DVFS) is the main technique to reduce the power dissipation of active systems. The convexity of the dynamic power curve with respect to clock frequency suggests that one can obtain better energy efficiency by reducing the frequency (coupled with the voltage) of the hardware components with a sacrifice from the execution time. Thus, a null clock frequency and infinite execution time would be the most energy efficient configuration when the dynamic part of the power is considered. However, the static part of the power eliminates this option as it might cancel out the gains with the extended execution time.

Time complexity models have facilitated the design of efficient algorithms. Similarly, power models can be a crucial step towards energy efficient algorithms for multi-core systems. Combined with the performance models, they can also shed light on the previously mentioned optimization problems.

In this thesis, we present a power model for multi-core systems and show how to integrate it with our throughput model to obtain energy efficiency (energy consumption per operation) of lock-free queue designs. Also, we have validated the power model with a more extensive set of lock-free data structures but these results [46–49] are not included in the thesis.

1.3.4 Execution Models and Analyses

It is common to model asynchronous executions by assuming an adversarial scheduler whose capabilities can vary depending on the context of study. This approach is convenient for the impossibility results and reasoning about the correctness properties of algorithms [4], but it leads to worst case bounds when the contention, connected to this performance, is analyzed. In the literature, complexity models have been proposed for contention in asynchronous shared memory systems. In [50], stall time, that is induced by memory operations that access to the same memory location at the same time interval (harware conflicts), is analyzed by assuming an adversarial scheduler. Both retry loop and hardware conflicts are considered in [51]. To capture the cost of contention, the total amount of work is bounded for an n-process lock-free update protocol where a process successfully updates a location once and returns from the protocol. In this study, the impact of exponential back-off is also analyzed.

In addition, amortized analysis techniques have been exploited [27, 43] to address the concurrency-related issues since the execution time of an individual lock-free operation cannot be bounded by definition. The failed attempts in the retry loops can be amortized by the successful ones, due to the fundamental property which states that a failed retry implies a successful concurrent retry. These analysis parameters can be set with a measure of contention to bound the average time complexity of the successful operations. Some common contention measures are:

- *Point Contention* [52]: maximum number of operations that are executed concurrently at any point during the execution interval of the operation
- *Interval Contention* [53]: number of operations whose execution interval overlaps with the execution interval of a given operation

The contention measure (that frames retry loop and hardware conflicts under a single contention cost) is often bounded by considering the worst case. However, the worst-case behavior is not enough to express the performance that we observe in practice. A tighter estimate of contention is needed because

the worst case is reached only if the concurrent operations access the same part of the data structure at the same time.

Close to the practical domain, the *expected* system and individual operation latencies are analyzed for a general class of lock-free algorithms under a uniform stochastic scheduler [21].

These theoretical analyses for the time complexity of lock-free data structures target the asymptotic behaviors in terms of number of processes. Also, empirical studies [54, 55] have been conducted to understand the throughput and energy efficiency. These empirical studies help to grasp the complicated interaction between software and hardware. However, there is a lack of analytical results that target the performance of lock-free data structures, that is observed in practice, with the consideration of the underlying hardware. This thesis aims to bridge the gap between theoretical bounds and actual measured performance.

In this thesis, we model and analyze the performance and energy efficiency of lock-free data structures on top of real hardware platforms. The modeling phase transforms the system, that constitutes lock-free program and machine, into an execution model, and the analysis of the model yields numeric values for the metrics of interest (*e.g.* throughput, cache misses, energy efficiency). This process is iterated throughout this thesis to tackle different types of lock-free data structures and different use cases, in which impacting factors might vary.

We start the process with the abstractions of the lock-free program and the machine that are characterized by a set of parameters. Then, the system is mapped to an execution model (*e.g.* cyclic pattern, Markov chain, Poisson process, queueing model in steady states under low and high contention, a system of mathematical equations) which retains the initial parameters. Both of these steps are aligned with the identified, significant performance impacting factors because we aim at representing the actual behavior of the system under a reasonable model complexity. During this process, we might ignore memory management calls if they are not costly, some type of hardware or algorithmic conflicts, and events when they are improbable. We collect the evidence regarding the insignificance of these details through empirical observations (benchmarking, performance counters). In a second phase, we analyze the execution model to

estimate (or sometimes bound) the main performance metrics: throughput and power consumption, that can be merged to obtain the energy efficiency. Finally, we validate our models with both synthetic tests and examples picked from application domains, for a range of lock-free data structures. To the best of our knowledge, we attempt for the first time to model and analyze the performance of lock-free data structures on such a broad domain and obtain estimates that are close to what is observed in practice.

An analytical framework can be useful in many ways. In the first place, it can explain observations and provide an understanding of the phenomena that drive the performance of lock-free data structures. It can identify the issues and bottlenecks in a design which in turn facilitates design decisions.

Secondly, it can be used to rank alternative lock-free data structure designs. We have mentioned in the previous sections that a vast variety of lock-free data structure designs exist. Different lock-free data structure designs can outperform each other in different configurations, which makes it difficult to conduct a fair comparison. Sometimes strengths or limitations of the data structures are hidden, thus unnoticeable even by their creators because they only appear in some configurations of the domain that it is often not possible to sufficiently cover empirically. An analytical framework can reveal the merits of data structures and provide a fair comparison by covering the whole configuration domain.

Last but not least, it can help the tuning process of the data structure related parameters. On this last point, lock-free data structures come with specific parameters, *e.g.* back-off, padding, and memory management related parameters, and become competitive only after picking carefully their values, which often involves a costly brute force approach. This can be replaced, or at least driven, by an analytical estimation of the performance.

1.4 Contributions

This thesis proposes analytical approaches to model and analyze the throughput and energy consumption of concurrent lock-free data structures. The contribu-

tions of this thesis to the field of concurrent lock-free data structures can be summarized under three headers:

Performance of a general class of lock-free data structures (Paper I and II): We present new ways of modeling and analyzing the performance of a general class of lock-free algorithms. We rely on the universal construction [9] and use this basic structure that is based on a single retry loop to model the lock-free data structures. A sequence of operations that are interleaved by application specific code abstracts the usage pattern of a lock-free data structure, by a thread. In this context, the performance metric (throughput) is defined as the average number of successful operations on the data structure per unit of time, by any thread.

We emphasize two impacting factors that rule the performance: (i) *stall time* due to the serialization of atomic primitives; (ii) *number of failed retry loop iterations*. We analyze these factors through a set of hardware and algorithmic parameters, and the impact of the latency of application specific work (which regulates the access frequency of the threads to the data structure) is underlined. We propose three analytical frameworks. We first target the cases where the latency of application specific code is a constant and we address it with a deterministic model (Paper I). The last two frameworks are based on stochastic models (Paper II). On the one hand, we address accurately cases where the latency is instantiated with exponential distribution through an execution model relying on Markov chains. On the other hand, we provide a generic approach that can be used for any latency distribution thanks to an approach based on queueing theory results. In addition, we exploit our frameworks to design a new back-off mechanism, to optimize memory management related parameters and to compare different lock-free data structures while covering the whole contention domain.

Performance and Energy Efficiency of Lock-Free Queues (Paper III): Our main contribution is to provide a generic high-level model for the performance and power dissipation of applications that rely heavily on the utilization of concurrent queues.

We have already mentioned that any sequential data structure can be trans-

formed into a lock-free linearizable concurrent data structure based on a single retry loop that applies the whole operation with a single successful atomic primitive. However, this approach might be far from being practical for some data structures. Due to performance concerns, lock-free data structure operations often possess more intricate designs. In this study, we want to have a broader relevance when dealing with these intricate operation designs and when threads are allowed to execute these different types of operations concurrently. We rely on an abstract model and calibrate it with samples from the solution space. This approach contrasts with the previously mentioned studies (Paper I and II) in which the execution models are more refined, and no samples from the solution space are used. We study lock-free queues and examine the interference between operations through the state of the queue (mostly empty or not empty). Based on their local and global interaction, we model the dequeuers' throughput and enqueuers' throughput focusing on the possible steady-state behaviors.

To model power dissipation, we first split the total power into *static, activation* and *dynamic* parts, the latter only depending on the actual instructions being executed. We further decompose these parts according to the hardware components (*memory, CPU, uncore*) and characterize their power consumption based on the rate of hardware events.

We instantiate our models using a very limited amount of application specific information, thanks to our performance model. Finally, we validate our models using several lock-free queue implementations through both synthetic tests and code from the application domain.

Performance of Lock-Free Search Data Structures (Paper IV): We study the throughput performance of concurrent lock-free search data structures. Search data structures possess different characteristics compared to the set of data structures that we studied formerly. In this study, we target the use cases where search data structures are utilized through a sequence of operations which are generated with a memoryless and stationary access pattern (*i.e.* for each operation in the sequence, the probability of selecting a specific key and a specific type are constants).

Search data structures are composed of basic blocks (nodes) that are linked

to each other in a way that provides efficient operations. Each thread executes a sequence of operations, and each operation triggers *read* and *modify* events on a subset of blocks. Hence, the throughput is ruled by the number of events in an operation and the latencies of these events.

The primary challenge in predicting throughput is that the latency of each event mainly depends on the state of the caches at the time when it is triggered. The state of caches is changing due to events that are triggered by the operations of multiple threads. Accordingly, the latency of an event is determined by the ordering of the events on the timeline. Considering a given block, two point distributions define the location of the events on the timeline. For each block, we use Poisson processes to model these point distributions relying on the properties of our access pattern and the rareness of events. Superposition and thinning properties of Poisson processes help us to deal with the interaction of threads. Knowing the probabilistic ordering of the events from single and multiple threads, we are able to estimate the throughput.

The validation of our model is conducted through several fundamental lock-free search data structures such as a hash table, linked list, skip list and binary tree. We rely then on this performance modeling to achieve performance optimization by analyzing the influence of possible memory alignment strategies. By aligning blocks to cache lines, the false sharing possibility can be eliminated at the expense of increasing the memory footprint of the search data structure. In this study, we also shed light on this trade-off in the context of search data structures.

Bibliography

[1] Gordon Earle Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, pp. 82–85, 1998.

[2] Geoff Lowney, "Why intel is designing multi-core processors," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2006, p. 113, ACM.

[3] Aaron Turon, *Understanding and Expressing Scalable Concurrency*, Ph.D. thesis, College of Computer and Information Science, Northeastern University, 2013.

[4] Maurice Herlihy and Nir Shavit, *The art of multiprocessor programming*, Morgan Kaufmann, 2008.

[5] Maurice Herlihy and Jeannette M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

[6] Leslie Lamport, "How to make a correct multiprocess program execute correctly on a multiprocessor," *IEEE Transactions on Computers (TC)*, vol. 46, no. 7, pp. 779–782, 1997.

[7] Leslie Lamport, "The mutual exclusion problem: partii - statement and solutions," *Journal of the ACM (JACM)*, vol. 33, no. 2, pp. 327–348, 1986.

[8] Maurice Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 1, pp. 124–149, 1991.

[9] Maurice Herlihy, "A methodology for implementing highly concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 15, no. 5, pp. 745–770, 1993.

[10] Maurice Herlihy, Victor Luchangco, and Mark Moir, "Obstruction-free synchronization: Double-ended queues as an example," in *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2003, pp. 522–529, IEEE Computer Society.

[11] "Intel's threading building blocks framework," `https://www.threadingbuildingblocks.org/`, Accessed: 2016-01-20.

[12] "Java concurrency package," `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html`, Accessed: 2016-01-20.

[13] "Microsoft .net framework," `http://www.microsoft.com/net`, Accessed: 2016-01-20.

[14] Leslie Lamport, "A new solution of dijkstra's concurrent programming problem," *Communications of the ACM*, vol. 17, no. 8, pp. 453–455, 1974.

[15] James Aspnes, *Wait-Free Consensus*, Ph.D. thesis, Carnegie-Mellon University, 1992.

[16] John L. Hennessy and David A. Patterson, *Computer Architecture - A Quantitative Approach, 5th Edition*, Morgan Kaufmann, 2012.

[17] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 2013, pp. 33–48, ACM.

[18] Paul E. Mckenney, "Memory barriers: a hardware view for software hackers," 2009.

[19] Robert D. Blumofe and Charles E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.

[20] Anders Gidenstam, Håkan Sundell, and Philippas Tsigas, "Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency," in *Proceedings of the International Conference on Principle of Distributed Systems (OPODIS)*. 2010, pp. 302–317, Springer.

[21] Dan Alistarh, Keren Censor-Hillel, and Nir Shavit, "Are lock-free concurrent algorithms practically wait-free?," in *Proceedings of the ACM Symposium on Theory of Computing (STOC)*. 2014, pp. 714–723, ACM.

[22] Amos Israeli and Lihu Rappoport, "Disjoint-access-parallel implementations of strong shared memory primitives," in *Proceedings of the ACM Symposium on Principles of Distributed Computing (PoDC)*. 1994, pp. 151–160, ACM.

[23] R. Kent Treiber, *Systems programming: Coping with parallelism*, International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[24] Håkan Sundell and Philippas Tsigas, "Fast and lock-free concurrent priority queues for multi-thread systems," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 65, no. 5, pp. 609–627, 2005.

[25] Joel Gibson and Vincent Gramoli, "Why non-blocking operations should be selfish," in *Proceedings of the International Symposium on Distributed Computing (DISC)*. 2015, pp. 200–214, Springer.

[26] Keir Fraser, *Practical lock-freedom*, Ph.D. thesis, University of Cambridge, UK, 2004.

[27] Bapi Chatterjee, Nhan Nguyen Dang, and Philippas Tsigas, "Efficient lock-free binary search trees," in *Proceedings of the ACM Symposium on Principles of Distributed Computing (PoDC)*. 2014, pp. 322–331, ACM.

[28] Aravind Natarajan and Neeraj Mittal, "Fast concurrent lock-free binary search trees," in *Principles and Practice of Parallel Programming (PPoPP)*. 2014, pp. 317–328, ACM.

[29] D. Hendler, N. Shavit, and L. Yerushalmi, "A Scalable Lock-Free Stack Algorithm," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 70, no. 1, pp. 1–12, 2010.

[30] Danny Hendler, Nir Shavit, and Lena Yerushalmi, "A scalable lock-free stack algorithm," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 70, no. 1, pp. 1–12, 2010.

[31] Moshe Hoffman, Ori Shalev, and Nir Shavit, "The baskets queue," in *Proceedings of the International Conference on Principle of Distributed Systems (OPODIS)*. 2007, pp. 401–414, Springer.

[32] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit, "Using elimination to implement scalable and lock-free fifo queues," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2005, pp. 253–262, ACM.

[33] Philippas Tsigas and Yi Zhang, "A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2001, pp. 134–143, ACM.

[34] Maged M. Michael and Michael L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the ACM Symposium on Principles of Distributed Computing (PoDC)*. 1996, pp. 267–275, ACM.

[35] Ivan Walulya and Philippas Tsigas, "Scalable lock-free vector with combining," in *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*. 2017, pp. 917–926, IEEE Computer Society.

[36] Håkan Sundell, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas, "A lock-free algorithm for concurrent bags," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2011, pp. 335–344, ACM.

[37] Maged M. Michael, "Cas-based lock-free algorithm for shared deques," in *Euro-Par COnference*. 2003, pp. 651–660, Springer.

[38] Håkan Sundell and Philippas Tsigas, "Lock-free deques and doubly linked lists," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 68, no. 7, pp. 1008–1020, 2008.

[39] Jonatan Lindén and Bengt Jonsson, "A skiplist-based concurrent priority queue with minimal memory contention," in *Proceedings of the International Conference on Principle of Distributed Systems (OPODIS)*. 2013, pp. 206–220, Springer.

[40] Nir Shavit and Itay Lotan, "Skiplist-based concurrent priority queues," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. 2000, pp. 263–268, IEEE Computer Society.

[41] Nhan Nguyen and Philippas Tsigas, "Lock-free cuckoo hashing," in *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2014, pp. 627–636, IEEE Computer Society.

[42] Maged M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2002, pp. 73–82, ACM.

[43] Mikhail Fomitchev and Eric Ruppert, "Lock-free linked lists and skip lists," in *Proceedings of the ACM Symposium on Principles of Distributed Computing (PoDC)*. 2004, pp. 50–59, ACM.

[44] Timothy L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the International Symposium on Distributed Computing (DISC)*. 2001, vol. 2180 of *Lecture Notes in Computer Science*, pp. 300–314, Springer.

[45] Jack J. Dongarra and Peter H. Beckman, "The international exascale software roadmap," *International Journal of High Performance Computing Applications (IJHPCA)*, vol. 25, no. 1, pp. 3–60, 2011.

[46] Phuong Ha, Vi Ngoc-Nha Tran, Ibrahim Umar, Philippas Tsigas, Anders Gidenstam, Paul Renaud-Goud, Ivan Walulya, and Aras Atalar, "Models for energy consumption of data structures and algorithms," *CoRR*, vol. abs/1801.09992, 2018.

[47] Phuong Hoai Ha, Vi Ngoc-Nha Tran, Ibrahim Umar, Aras Atalar, Anders Gidenstam, Paul Renaud-Goud, and Philippas Tsigas, "White-box methodologies, programming abstractions and libraries," *CoRR*, vol. abs/1801.08761, 2018.

[48] Phuong Ha, Vi Ngoc-Nha Tran, Ibrahim Umar, Aras Atalar, Anders Gidenstam, Paul Renaud-Goud, Philippas Tsigas, and Ivan Walulya, "Power models, energy models and libraries for energy-efficient concurrent data structures and algorithms," *CoRR*, vol. abs/1801.10556, 2018.

[49] Phuong Hoai Ha, Vi Ngoc-Nha Tran, Ibrahim Umar, Aras Atalar, Anders Gidenstam, Paul Renaud-Goud, Philippas Tsigas, and Ivan Walulya, "D2.4 report on the final prototype of programming abstractions for energy-efficient inter-process communication," *CoRR*, vol. abs/1802.03013, 2018.

[50] Cynthia Dwork, Maurice Herlihy, and Orli Waarts, "Contention in shared memory algorithms," *Journal of the ACM (JACM)*, vol. 44, no. 6, pp. 779–805, 1997.

[51] Naama Ben-David and Guy E. Blelloch, "Analyzing contention and backoff in asynchronous shared memory," in *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, 2017, pp. 53–62.

[52] Hagit Attiya and Arie Fouren, "Algorithms adapting to point contention," *Journal of the ACM (JACM)*, vol. 50, no. 4, pp. 444–468, 2003.

[53] Yehuda Afek, Gideon Stupp, and Dan Touitou, "Long lived adaptive splitter and applications," *Journal of Distributed Computing*, vol. 15, no. 2, pp. 67–86, 2002.

[54] Vincent Gramoli, "More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms," in *Principles and Practice of Parallel Programming (PPoPP)*. 2015, pp. 1–10, ACM.

[55] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis, "Asynchronized concurrency: The secret to scaling concurrent search data structures," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2015, pp. 631–644, ACM.

Part II

RESULTS

RESULT I

Aras Atalar, Paul Renaud-Goud and Philippas Tsigas

Analyzing the Performance of Lock-Free Data Structures: A Conflict-Based Model

In the Proceedings of 29th International Symposium on Distributed Computing (DISC 2015)

pages 341-355, Springer-Verlag 2015.

2

RESULT I - Analyzing the Performance of Lock-Free Data Structures: A Conflict-Based Model

Abstract

This paper considers the modeling and the analysis of the performance of lock-free concurrent data structures. Lock-free designs employ an optimistic conflict control mechanism, allowing several processes to access the shared data object at the same time. They guarantee that at least one concurrent operation finishes in a finite number of its own steps regardless of the state of the operations. Our analysis considers such lock-free data structures that can be represented as linear combinations of fixed size retry loops.

Our main contribution is a new way of modeling and analyzing a general class of lock-free algorithms, achieving predictions of throughput that are close to what we observe in practice. We emphasize two kinds of conflicts that shape the performance: (i) hardware conflicts, due to concurrent calls to atomic primitives; (ii) logical conflicts, caused by simultaneous operations on the shared data structure.

We show how to deal with these hardware and logical conflicts separately, and how to combine them, so as to calculate the throughput of lock-free algorithms. We propose also a common framework that enables a fair comparison between lock-free implementations by covering the whole contention domain, together with a better understanding of the performance impacting factors. This part of our analysis comes with a method for calculating a good back-off strategy to finely tune the performance of a lock-free algorithm. Our experimental results, based on a set of widely used concurrent data structures and on abstract lock-free designs, show that our analysis follows closely the actual code behavior.

2.1 Introduction

Lock-free programming provides highly concurrent access to data and has been increasing its footprint in industrial settings. Providing a modeling and an analysis framework capable of describing the practical performance of lock-free algorithms is an essential, missing resource necessary to the parallel programming and algorithmic research communities in their effort to build on previous intellectual efforts. The definition of lock-freedom mainly guarantees that at least one concurrent operation on the data structure finishes in a finite number of its own steps, regardless of the state of the operations. On the individual operation level, lock-freedom cannot guarantee that an operation will not starve.

The goal of this paper is to provide a way to model and analyze the practically observed performance of lock-free data structures. In the literature, the common performance measure of a lock-free data structure is the throughput, *i.e.* the number of successful operations per unit of time. It is obtained while

threads are accessing the data structure according to an access pattern that interleaves local work between calls to consecutive operations on the data structure. Although this access pattern to the data structure is significant, there is no consensus in the literature on what access to be used when comparing two data structures. So, the amount of local work (that we will refer as parallel work for the rest of the paper) could be constant ([1, 2]), uniformly distributed ([3], [4]), exponentially distributed ([5], [6]), null ([7, 8]), *etc.* More questionably, the average amount is rarely scanned, which leads to a partial covering of the contention domain.

We propose here a common framework enabling a fair comparison between lock-free data structures, while exhibiting the main phenomena that drive performance, and particularly the contention, which leads to different kinds of conflicts. As this is the first step in this direction, we want to deeply analyze the core of the problem, without impacting factors being diluted within a probabilistic smoothing. Therefore, we choose a constant local work, hence constant access rate to the data structures. In addition to the prediction of the data structure performance, our model provides a good back-off strategy, that achieves the peak performance of a lock-free algorithm.

Two kinds of conflict appear during the execution of a lock-free algorithm, both of them leading to additional work. Hardware conflicts occur when concurrent operations call atomic primitives on the same memory location: these calls collide and conduct to stall time, that we name here *expansion*. Logical conflicts take place if concurrent operations overlap: because of the lock-free nature of the algorithm, several concurrent operations can run simultaneously, but usually only one retry can logically succeed. We show that the additional work produced by the failures is not necessarily harmful for the system-wise performance.

We then show how throughput can be computed by connecting these two key factors in an iterative way. We start by estimating the expansion probabilistically, and emulate the effect of stall time introduced by the hardware conflicts as extra work added to each thread. Then we estimate the number of failed operations, that in turn lead to additional extra work, by computing again the

expansion on a system setting where those two new amounts of work have been incorporated, and reiterate the process; the convergence is ensured by a fixed-point search.

We consider the class of lock-free algorithms that can be modeled as a linear composition of fixed size retry loops. This class covers numerous extensively used lock-free designs such as stacks [9] (Pop, Push), queues [1] (Enqueue, Dequeue), counters [4] (Increment, Decrement) and priority queues [8] (DeleteMin).

To evaluate the accuracy of our model and analysis framework, we performed experiments both on synthetic tests, that capture a wide range of possible abstract algorithmic designs, and on several reference implementations of extensively studied lock-free data structures. Our evaluation results reveal that our model is able to capture the behavior of all the synthetic and real designs for all different numbers of threads and sizes of parallel work (consequently also contention). We also evaluate the use of our analysis as a tool for tuning the performance of lock-free code by selecting the appropriate back-off strategy that will maximize throughput by comparing our method against widely known back-off policies, namely linear and exponential.

The rest of the paper is organized as follows. We discuss related work in Section 2.2, then the problem is formally described in Section 2.3. We consider the logical conflicts in the absence of hardware conflicts in Section 2.4. In Section 2.5, we firstly show how to compute the expansion, then combine hardware and logical conflicts to obtain the final throughput estimate. We describe the experimental results in Section 3.6.

2.2 Related Work

Anderson *et al.* [10] evaluated the performance of lock-free objects in a single processor real-time system by emphasizing the impact of retry loop interference. Tasks can be preempted during the retry loop execution, which can lead to interference, and consequently to an inflation in retry loop execution due to retries. They obtained upper bounds for the number of interferences under

various scheduling schemes for periodic real-time tasks.

Intel [11] conducted an empirical study to illustrate performance and scalability of locks. They showed that the critical section size, the time interval between releasing and re-acquiring the lock (that is similar to our parallel section size) and number of threads contending the lock are vital parameters.

Failed retries do not only lead to useless effort but also degrade the performance of successful ones by contending the shared resources. Alemany *et al.* [12] have pointed out this fact, that is in accordance with our two key factors, and, without trying to model it, have mitigated those effects by designing non-blocking algorithms with operating system support.

Alistarh *et al.* [13] have studied the same class of lock-free structures that we consider in this paper. The analysis is done in terms of scheduler steps, in a system where only one thread can be scheduled (and can then run) at each step. If compared with execution time, this is particularly appropriate to a system with a single processor and several threads, or to a system where the instructions of the threads cannot be done in parallel (*e.g.* multi-threaded program on a multi-core processor with only read and write on the same cache line of the shared memory). In our paper, the execution is evaluated in terms of processor cycles, strongly related to the execution time. In addition, the "parallel work" and the "critical work" can be done in parallel, and we only consider retry-loops with one Read and one CAS, which are serialized. In addition, they bound the asymptotic expected system latency (with a big O, when the number of threads tends to infinity), while in our paper we estimate the throughput (close to the inverse of system latency) for any number of threads.

2.3 Problem Statement

2.3.1 Running Program and Targeted Platform

In this paper, we aim at evaluating the throughput of a multi-threaded algorithm that is based on the utilization of a shared lock-free data structure that relies on a single retry loop which applies the whole operation with a single

```
Procedure AbstractAlgorithm
1 Initialization();
2 while ! done do
3 |   Parallel_Work();
4 |   while ! success do
5 |   |   current ← Read(AP);
6 |   |   new ← Critical_Work(current);
7 |   |   success ← CAS(AP, current, new);
```

Figure 2.1: Thread procedure

successful atomic primitive. Such a program can be abstracted by the Procedure AbstractAlgorithm (see Figure 3.1) that represents the skeleton of the function which is called by each spawned thread. It is decomposed in two main phases: the *parallel section*, represented on line 2, and the *retry loop*, from line 3 to line 6. A *retry* starts at line 4 and ends at line 6.

As for line 1, the function Initialization shall be seen as an abstraction of the delay between the spawns of the threads, that is expected not to be null, even when a barrier is used. We then consider that the threads begin at the exact same time, but have different initialization times.

The parallel section is the part of the code where the thread does not access the shared data structure; the work that is performed inside this parallel section can possibly depend on the value that has been read from the data structure, *e.g.* in the case of processing an element that has been dequeued from a FIFO (First-In-First-Out) queue.

In each retry, a thread tries to modify the data structure, and does not exit the retry loop until it has successfully modified the data structure. It does that by firstly reading the access point AP of the data structure, then according to the value that has been read, and possibly to other previous computations that occurred in the past, the thread prepares the new desired value as an access point of the data structure. Finally, it atomically tries to perform the change

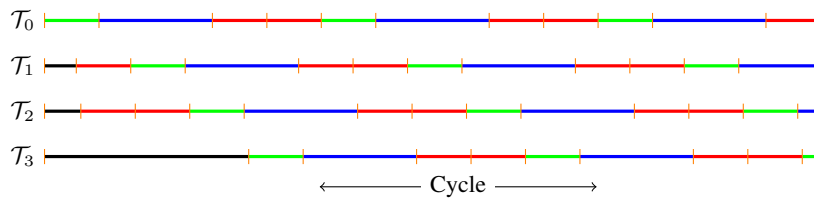


Figure 2.2: Execution with one wasted retry, and one inevitable failure

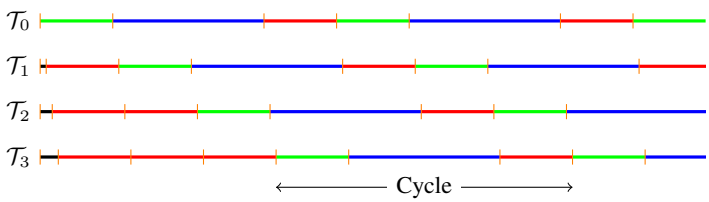


Figure 2.3: Execution with minimum number of failures

through a call to the *Compare-And-Swap* (*CAS*) primitive. If it succeeds, *i.e.* if the access point has not been changed by another thread between the first *Read* and the *CAS*, then it goes to the next parallel section, otherwise it repeats the process. The retry loop is composed of at least one retry, and we number the retries starting from 0, since the first iteration of the retry loop is actually not a retry, but a try.

We analyze the behavior of AbstractAlgorithm from a throughput perspective, which is defined as the number of successful data structure operations per unit of time. In the context of Procedure AbstractAlgorithm, it is equivalent to the number of successful *CAS*s.

The throughput of the lock-free algorithm, that we denote by T, is impacted by several parameters.

- *Algorithm parameters*: the amount of work inside a call to Parallel_Work (resp. Critical_Work) denoted by pw (resp. cw).
- *Platform parameters*: *Read* and *CAS* latencies (rc and cc respectively),

and the number P of processing units (cores). We assume homogeneity for the latencies, *i.e.* every thread experiences the same latency when accessing an uncontended shared data, which is achieved in practice by pinning threads to the same socket.

2.3.2 Examples and Issues

We first present two straightforward upper bounds on the throughput, and describe the two kinds of conflict that keep the actual throughput away from those upper bounds.

2.3.2.1 Immediate Upper Bounds

Trivially, the minimum amount of work $rlw^{(-)}$ in a given retry is $rlw^{(-)} = rc + cw + cc$, as we should pay at least the memory accesses and the critical work cw in between.

Thread-wise: A given thread can at most perform one successful retry every $pw + rlw^{(-)}$ units of time. In the best case, P threads can then lead to a throughput of $P/(pw + rlw^{(-)})$.

System-wise: By definition, two successful retries cannot overlap, hence we have at most 1 successful retry every $rlw^{(-)}$ units of time.

Altogether, the throughput T is bounded by

$$T \leq \min\left(\frac{1}{rc + cw + cc}, \frac{P}{pw + rc + cw + cc}\right), \textit{i.e.}$$

$$T \leq \begin{cases} \frac{1}{rc+cw+cc} & \text{if } pw \leq (P-1)(rc + cw + cc) \\ \frac{P}{pw+rc+cw+cc} & \text{otherwise.} \end{cases} \tag{2.1}$$

2.3.2.2 Conflicts

Logical conflicts Equation 2.1 expresses the fact that when pw is small enough, *i.e.* when $pw \leq (P-1)rlw^{(-)}$, we cannot expect that every thread performs a

successful retry every $pw + rlw^{(-)}$ units of time, since it is more than what the retry loop can afford. As a result, some logical conflicts, hence unsuccessful retries, will be inevitable, while the others, if any, are called *wasted*.

However, different executions can lead to different numbers of failures, which end up with different throughput values. Figures 2.2 and 2.3 depict two executions, where the black parts are the calls to Initialization, the blue parts are the parallel sections, and the retries can be either unsuccessful — in red — or successful — in green. We experiment different initialization times, and observe different synchronizations, hence different numbers of wasted retries. After the initial transient state, the execution depicted in Figure 2.3 comprises only the inevitable unsuccessful retries, while the execution of Figure 2.2 contains one wasted retry.

We can see on those two examples that a cyclic execution is reached after the transient behavior; actually, we show in Section 2.4 that, in the absence of hardware conflicts, every execution will become periodic, if the initialization times are spaced enough. In addition, we prove that the shortest period is such that, during this period, every thread succeeds exactly once. This finally leads us to define the additional failures as wasted, since we can directly link the throughput with this number of wasted retries: a higher number of wasted retries implying a lower throughput.

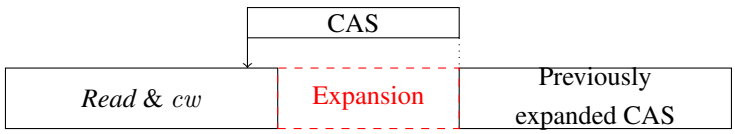


Figure 2.4: Expansion

Hardware conflicts The requirement of atomicity compels the ownership of the data in an exclusive manner by the executing core. This fact prohibits concurrent execution of atomic instructions if they are operating on the same data. Therefore, overlapping parts of atomic instructions are serialized by the hard-

ware, leading to stalls in subsequently issued ones. For our target lock-free algorithm, these stalls that we refer to as expansion become an important slowdown factor in case threads interfere in the retry loop. As illustrated in Figure 2.4, the latency for *CAS* can expand and cause remarkable decreases in throughput since the *CAS* of a successful thread is then expanded by others; for this reason, the amount of work inside a retry is not constant, but is, generally speaking, a function depending on the number of threads that are inside the retry loop.

2.3.2.3 Process

We deal with the two kinds of conflicts separately and connect them together through the fixed-point iterative convergence.

In Section 2.5.1, we compute the expansion in execution time of a retry, noted e, by following a probabilistic approach. The estimation takes as input the expected number of threads inside the retry loop at any time, and returns the expected increase in the execution time of a retry due to the serialization of atomic primitives.

In Section 2.4, we are given a program without hardware conflicts described by the size of the parallel section $pw^{(+)}$ and the size of a retry $rlw^{(+)}$. We compute upper and lower bounds on the throughput T, the number of wasted retries w, and the average number of threads inside the retry loop P_{rl}. Without loss of generality, we can normalize those execution times by the execution time of a retry, and define the parallel section size as $pw^{(+)} = q + r$, where q is a non-negative integer and r is such that $0 \leq r < 1$. This pair (together with the number of threads P) constitutes the actual input of the estimation.

Finally, we combine those two outcomes in Section 2.5.2 by emulating expansion through work not prone to hardware conflicts and obtain the full estimation of the throughput.

2.4 Execution without hardware conflict

We show in this section that, in the absence of hardware conflicts, the execution becomes periodic, which eases the calculation of the throughput. We start by defining some useful concepts: (f, P)-cyclic executions are special kind of periodic executions such that within the shortest period, each thread performs exactly f unsuccessful retries and 1 successful retry. The *well-formed seed* is a set of events that allows us to detect an (f, P)-*cyclic execution* early, and the *gaps* are a measure of the quality of the synchronization between threads. The idea is to iteratively add threads into the game and show that the periodicity is maintained. Theorem 1 establishes a fundamental relation between gaps and well-formed seeds, while Theorem 2 proves the periodicity, relying on the disjoint cases of Lemma 2, 3, and 4. Finally, we exhibit upper and lower bounds on throughput and number of failures, along with the average number of threads inside the retry loop.

2.4.1 Setting

2.4.1.1 Initial Restrictions

Remark 1. Concerning correctness, we assume that the reference point of the *Read* and the *CAS* occurs when the thread enters and exits any retry, respectively.

Remark 2. We do not consider simultaneous events, so all inequalities that refer to time comparison are strict, and can be viewed as follows: time instants are real numbers, and can be equal, but every event is associated with a thread; also, in order to obtain a strict order relation, we break ties according to the thread numbers (for instance with the relation $<$).

2.4.1.2 Notations and Definitions

We recall that P threads are executing the pseudo-code described in Procedure AbstractAlgorithm, one retry is of unit-size, and the parallel section is of size $pw^{(+)} = q + r$, where q is a non-negative integer and r is such that

$0 \leq r < 1$. Considering a thread $\mathcal{T}_n$ which succeeds at time S_n; this thread completes a whole retry in 1 unit of time, then executes the parallel section of size $pw^{(+)}$, and attempts to perform again the operation every unit of time, until one of the attempt is successful.

Definition 1. *An execution with P threads is called (C, P)-cyclic execution if and only if (i) the execution is periodic, i.e. at every time, every thread is in the same state as one period before, (ii) the shortest period contains exactly one successful attempt per thread, (iii) the shortest period is $1 + q + r + C$.*

Definition 2. *Let $\mathcal{S} = (\mathcal{T}_i, S_i)_{i \in [\![0, P-1]\!]}$, where $\mathcal{T}_i$ are threads and S_i ordered times, i.e. such that $S_0 < S_1 < \cdots < S_{P-1}$. $\mathcal{S}$ is a seed if and only if for all $i \in [\![0, P-1]\!]$, $\mathcal{T}_i$ does not succeed between S_0 and S_i, and starts a retry at S_i.*

We define $f(\mathcal{S})$ as the smallest non-negative integer such that $S_0 + 1 + q + r + f(\mathcal{S}) > S_{P-1} + 1$, i.e. $f(\mathcal{S}) = \max(0, \lceil S_{P-1} - S_0 - q - r \rceil)$. When $\mathcal{S}$ is clear from the context, we denote $f(\mathcal{S})$ by f.

Definition 3. *$\mathcal{S}$ is a well-formed seed if and only if for each $i \in [\![0, P-1]\!]$, the execution of thread $\mathcal{T}_i$ contains the following sequence: a successful retry starting at S_i, the parallel section, f unsuccessful retries, then a successful retry.*

Those definitions are coupled through the two natural following properties:

Property 1. *Given a (C, P)-cyclic execution, any seed $\mathcal{S}$ including P consecutive successes is a well-formed seed, with $f(\mathcal{S}) = C$.*

Proof. Choosing any set of P consecutive successes, we are ensured, by the definition of a (f, P)-cyclic execution, that for each thread, after the first success, the next success will be obtained after f failures. The order will be preserved, and this shows that a seed including our set of successes is actually a well-formed seed. □

Property 2. *If there exists a well-formed seed in an execution, then after each thread succeeded once, the execution coincides with an (f, P)-cyclic execution.*

Proof. By the definition of a well-formed seed, we know that the threads will first succeed in order, fails f times, and succeed again in the same order. Considering the second set of successes in a new well-formed seed, we observe that the threads will succeed a third time in the same order, after failing f times. By induction, the execution coincides with an (f, P)-cyclic execution. □

Together with the seed concept, we define the notion of *gap* that we will use extensively in the next subsection. The general idea of those gaps is that within an (f, P)-cyclic execution, the period is higher than $P \times 1$, which is the total execution time of all the successful retries within the period. The difference between the period (that lasts $1 + q + r + f$) and P, reduced by r (so that we obtain an integer), is referred as *lagging time* in the following. If the threads are numbered according to their order of success (modulo P), as the time elapsed between the successes of two given consecutive threads is constant (during the next period, this time will remain the same), this lagging time can be seen in a circular manner (see Figure 2.5): the threads are represented on a circle whose length is the lagging time increased by r, and the length between two consecutive threads is the time between the end of the successful retry of the first thread and the start of the successful retry of the second one. More formally, for all $(n, k) \in [\![0, P-1]\!]^2$, we define the gap $G_n^{(k)}$ between $\mathcal{T}_n$ and its k^{th} predecessor based on the gap with the first predecessor:

$$\begin{cases} \forall n \in [\![1, P-1]\!] \quad ; \quad G_n^{(1)} = S_n - S_{n-1} - 1 \\ G_0^{(1)} = S_0 + q + r + f - S_{P-1} \end{cases},$$

which leads to the definition of higher order gaps:

$$\forall n \in [\![0, P-1]\!] \quad ; \quad \forall k > 0 \quad ; \quad G_n^{(k)} = \sum_{j=n-k+1}^{n} G_{j \bmod P}^{(1)}.$$

For consistency, for all $n \in [\![0, P-1]\!]$, $G_n^{(0)} = 0$.

Equally, the gaps can be obtained directly from the successes: for all $k \in [\![1, P-1]\!]$,

$$G_n^{(k)} = \begin{cases} S_n - S_{n-k} - k & \text{if } n > k \\ S_n - S_{P+n-k} + 1 + q + r + f - k & \text{otherwise} \end{cases} \tag{2.2}$$

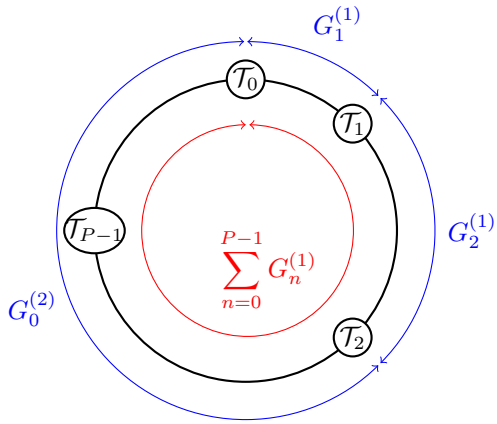


Figure 2.5: Gaps

Note that, in an (f, P)-cyclic execution, the lagging time is the sum of all first order gaps, reduced by r.

Now we extend the concept of well-formed seed to weakly-formed seed.

Definition 4. *Let $\mathcal{S} = (\mathcal{T}_i, S_i)_{i\in[\![0,P-1]\!]}$ be a seed.*

$\mathcal{S}$ is a weakly-formed seed for P threads if and only if: $(\mathcal{T}_i, S_i)_{i\in[\![0,P-2]\!]}$ is a well-formed seed for $P-1$ threads, and the first thread succeeding after $\mathcal{T}_{P-2}$ is $\mathcal{T}_{P-1}$.

Property 3. *Let $\mathcal{S} = (\mathcal{T}_i, S_i)_{i\in[\![0,P-1]\!]}$ be a weakly-formed seed.*

Denoting $f = f\left((\mathcal{T}_i, S_i)_{i\in[\![0,P-2]\!]}\right)$, for each $n \in [\![0, P-1]\!]$, $G_n^{(f)} < 1$.

Proof. We have $S_{P-2} + 1 < S_{P-1} < R_0^f$, and if we note indeed $\widetilde{G}_n^{(k)}$ the gaps within $(\mathcal{T}_i, S_i)_{i\in[\![0,P-2]\!]}$, the previous well-formed seed with $P-1$ threads, we know that for all $n \in [\![1, P-2]\!]$, $\widetilde{G}_n^{(1)} = G_n^{(1)}$, and $G_{P-1}^{(1)} + G_0^{(1)} = \widetilde{G}_0^{(1)}$, which leads to $G_n^{(k)} \leq \widetilde{G}_n^{(k)}$, for all $n \in [\![0, P-1]\!]$ and k; hence the weaker property. □

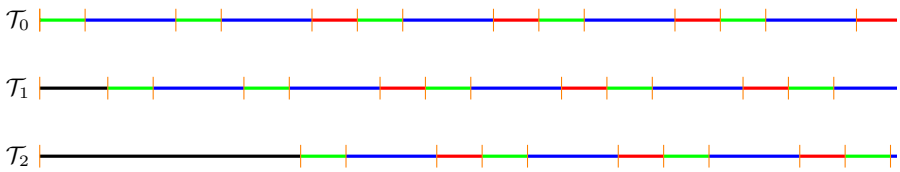


Figure 2.6: Lemma 2 configuration

2.4.2 Cyclic Executions

Theorem 1. *Given a seed $\mathcal{S} = (\mathcal{T}_i, S_i)_{i\in[\![0,P-1]\!]}$, $\mathcal{S}$ is a well-formed seed if and only if for all $n \in [\![0, P-1]\!]$, $0 \leq G_n^{(f)} < 1$.*

Proof. Let $\mathcal{S} = (\mathcal{T}_i, S_i)_{i\in[\![0,P-1]\!]}$ be a seed.
($\Leftarrow$) We assume that for all $n \in [\![0, P-1]\!]$, $0 < G_n^{(f)} < 1$, and we first show that the first successes occur in the following order: $\mathcal{T}_0$ at S_0, $\mathcal{T}_1$ at S_1, ..., $\mathcal{T}_{P-1}$ at S_{P-1}, $\mathcal{T}_0$ again at R_0^f. The first threads that are successful executes their parallel section after their success, then enters their second retry loop: from this moment, they can make the first attempt of the threads, that has not been successful yet, fail. Therefore, we will look at which retry of which already successful threads could have an impact on which other threads.

We can notice that for all $n \in [\![0, P-1]\!]$, if the first success of $\mathcal{T}_n$ occurs at S_n, then its next attempts will potentially occur at $R_n^k = S_n + 1 + q + r + k$, where $k \geq 0$. More specifically, thanks to Equation 2.2, for all $n \leq f$, $R_n^k = S_{P+n-f} + G_n^{(f)} + k$. Also, for all $k \leq f - n$,

$$\begin{aligned} R_n^k - S_{P+n-f+k} &= -\left(S_{P+n-f+k} - S_{P+n-f} - k\right) + G_n^{(f)} \\ &= G_n^{(f)} - G_{P+n-f+k}^{(k)} \\ R_n^k - S_{P+n-f+k} &= G_n^{(f-k)}, \end{aligned} \tag{2.3}$$

and this implies that if $k > 0$,

$$S_{P+n-f+k} - R_n^{k-1} = 1 - G_n^{(f-k)}. \tag{2.4}$$

We know, by hypothesis, that $0 < G_n^{(f-k)} < 1$, equivalently $0 < 1 - G_n^{(f-k)} < 1$. Therefore Equation 2.3 states that if a thread $\mathcal{T}_{n'}$ starts a successful attempt at $S_{P+n-f+k}$, then this thread will make the k^{th} retry of $\mathcal{T}_n$ fail, since $\mathcal{T}_n$ enters a retry while $\mathcal{T}_{n'}$ is in a successful retry. And Equation 2.4 shows that, given a thread $\mathcal{T}_{n'}$ starting a new retry at $S_{P+n-f+k}$, the only retry of $\mathcal{T}_n$ that can make $\mathcal{T}_{n'}$ fail on its attempt is the $(k-1)^{\text{th}}$ one. There is indeed only one retry of $\mathcal{T}_n$ that can enter a retry before the entrance of $\mathcal{T}_{n'}$, and exit the retry after it.

$\mathcal{T}_0$ is the first thread to succeed at S_0, because no other thread is in the retry loop at this time. Its next attempt will occur at R_0^0, and all thread attempts that start before S_{P-f} (included) cannot fail because of $\mathcal{T}_0$, since it runs then the parallel section. Also, since all gaps are positive, the threads $\mathcal{T}_1$ to $\mathcal{T}_{P-f}$ will succeed in this order, respectively starting at times S_1 to S_{P-f}.

Then, using induction, we can show that $\mathcal{T}_{P-f+1}, \ldots, \mathcal{T}_{P-1}$ succeed in this order, respectively starting at times $S_{P-f+1}, \ldots, S_{P-1}$. For $j \in [\![0, f-1]\!]$, let $(\mathcal{P}_j)$ be the following property: for all $n \in [\![0, P-f+j]\!]$, $\mathcal{T}_n$ starts a successful retry at S_n. We assume that for a given j, $(\mathcal{P}_j)$ is true, and we show that it implies that $\mathcal{T}_{P-f+j+1}$ will succeed at $S_{P-f+j+1}$. The successful attempt of $\mathcal{T}_{P-f+j}$ at S_{P-f+j} leads, for all $j' \in [\![0, j]\!]$, to the failure of the j'^{th} retry of $\mathcal{T}_{j-j'}$ (explanation of Equation 2.3). But for each $\mathcal{T}_{j'}$, this attempt was precisely the one that could have made $\mathcal{T}_{P-f+j+1}$ fail on its attempt at $S_{P-f+j+1}$ (explanation of Equation 2.3). Given that all threads $\mathcal{T}_n$, where $n > P-f+j+1$, do not start any retry loop before $S_{P-f+j+1}$, $\mathcal{T}_{P-f+j+1}$ will succeed at $S_{P-f+j+1}$. By induction, $(\mathcal{P}_j)$ is true for all $j \in [\![0, f-1]\!]$.

Finally, when $\mathcal{T}_{P-1}$ succeeds, it makes the $(f-1-n)^{\text{th}}$ retry of $\mathcal{T}_n$ fail, for all $n \in [\![0, f-1]\!]$; also the next potentially successful attempt for $\mathcal{T}_n$ is at R_n^{f-n}. (Naturally, for all $n \in [\![f, P-1]\!]$, the next potentially successful attempt for $\mathcal{T}_n$ is at R_n^0.)

We can observe that for all $n < P$, $j \in [\![0, P-1-n]\!]$, and all $k \geq j$,

$$\begin{aligned} R_{n+j}^{k-j} - R_n^k &= S_{n+j} + k - j - (S_n + k) \\ R_{n+j}^{k-j} - R_n^k &= G_{n+j}^{(j)}, \end{aligned} \tag{2.5}$$

hence for all $n \in [\![1, f]\!]$, $R_n^{f-n} - R_0^f = G_n^{(n)} > 0$.

$$R_n^{f-n} - R_0^f = G_n^{(n)} > 0.$$

As we have as well, for all $n \in [\![f+1, P-1]\!]$, $R_n^0 > R_f^0$, we obtain that among all the threads, the earliest possibly successful attempt is R_0^f. Following $\mathcal{T}_{P-1}$, $\mathcal{T}_0$ is consequently the next successful thread in its f^{th} retry.

To conclude this part, we can renumber the threads ($\mathcal{T}_{n+1}$ becoming now $\mathcal{T}_n$ if $n > 0$, and $\mathcal{T}_0$ becoming $\mathcal{T}_{P-1}$), and follow the same line of reasoning. The only difference is the fact that $\mathcal{T}_{P-1}$ (according to the new numbering) enters the retry loop f units of time before S_{P-1}, but it does not interfere with the other threads, since we know that those attempts will fail.

There remains the case where there exists $n \in [\![0, P-1]\!]$ such that $G_n^{(f)} = 0$. This implies that $f = 0$, thus we have a well-formed seed.

($\Rightarrow$) We prove now the implication by contraposition; we assume that there exists $n \in [\![0, P-1]\!]$ such that $G_n^{(f)} > 1$ or $G_n^{(f)} < 0$, and show that $\mathcal{S}$ is not a well-formed seed.

We assume first that an f^{th} order gap is negative. As it is a sum of 1^{st} order gaps, then there exists n' such that $G_{n'}^{(1)}$ is negative; let n'' be the highest one.

If $n'' > 0$, then either the threads $\mathcal{T}_0, \ldots, \mathcal{T}_{n''-1}$ succeeded in order at their 0^{th} retry, and then $\mathcal{T}_{n''-1}$ makes $\mathcal{T}_{n''}$ fail at its 0^{th} retry (we have a seed, hence by definition, $S_{n''-1} < S_{n''}$, and $G_{n''}^{(1)} < 0$, thus $S_{n''-1} < S_{n''} < S_{n''-1} + 1$), or they did not succeed in order at their first try. In both cases, $\mathcal{S}$ is not a well-formed seed.

If $n'' = 0$, let us assume that $\mathcal{S}$ is a well-formed seed. Let also a new seed be $\mathcal{S}' = (\mathcal{T}_i, S'_i)_{i \in [\![0, P-1]\!]}$, where for all $n \in [\![0, P-2]\!]$, $S'_{n+1} = S_n$, and $S'_0 = S_{P-1} - (q + 1 + f + r)$. Like $\mathcal{S}$, $\mathcal{S}'$ is a well-formed seed; however, $G_1^{(1)}$ is negative, and we fall back into the previous case, which shows that $\mathcal{S}'$ is not a well-formed seed. This is absurd, hence $\mathcal{S}$ is not a well-formed seed.

We assume now that every gap is positive and choose n_0 defined by: $n_0 = \min\{n \ ; \ \exists k \in [\![0, P-1]\!] / G_{n+k}^{(k)} > 1\}$, and $f_0 = \min\{k \ ; \ G_{n_0+k}^{(k)} > 1\}$:

among the gaps that exceed 1, we pick those that concern the earliest thread, and among them the one with the lowest order.

Let us assume that threads $\mathcal{T}_0, \ldots, \mathcal{T}_{P-1}$ succeed at their 0^{th} retry in this order, then $\mathcal{T}_0, \ldots, \mathcal{T}_{n_0}$ complete their second successful retry loop at their f^{th} retry, in this order. If this is not the case, then $\mathcal{S}$ is not a well-formed seed, and the proof is completed. According to Equation 2.5, we have, on the one hand, $R^{f_0-1}_{n_0+1} - R^{f_0}_{n_0} = G^{(1)}_{n_0+1}$, which implies $R^{f_0}_{n_0+1} - 1 - R^{f_0}_{n_0} = G^{(1)}_{n_0+1}$, thus $R^{f}_{n_0+1} - (R^{f}_{n_0} + 1) = G^{(1)}_{n_0+1}$; and on the other hand, $R^{0}_{n_0+f_0} - R^{f_0}_{n_0} = G^{(f_0)}_{n_0+f_0}$ implying $R^{f-f_0}_{n_0+f_0} - \big(R^{f}_{n_0} + 1\big) = G^{(f_0)}_{n_0+f_0} - 1$. As we know that $G^{(f_0)}_{n_0+f_0} - G^{(1)}_{n_0+1} = G^{(f_0-1)}_{n_0+f_0} < 1$ by definition of f_0 (and n_0), we can derive that $R^{f}_{n_0+1} - (R^{f}_{n_0} + 1) > R^{f-f_0}_{n_0+f_0} - (R^{f}_{n_0} + 1)$. We have assumed that $\mathcal{T}_{n_0}$ succeeds at its f^{th} retry, which will end at $R^{f}_{n_0} + 1$. The previous inequality states then that $\mathcal{T}_{n_0+1}$ cannot be successful at its f^{th} retry, since either a thread succeeds before $\mathcal{T}_{n_0+f_0}$ and makes both $\mathcal{T}_{n_0+f_0}$ and $\mathcal{T}_{n_0+1}$ fail, or $\mathcal{T}_{n_0+f_0}$ succeeds and makes $\mathcal{T}_{n_0+1}$ fail. We have shown that $\mathcal{S}$ is not a well-formed seed. □

Lemma 1. *Assuming $r \neq 0$, if a new thread is added to an (f, P)-cyclic execution, it will eventually succeed.*

Proof. Let R^0_P be the time of the 0^{th} retry of the new thread, that we number $\mathcal{T}_P$. If this retry is successful, we are done; let us assume now that this retry is a failure, and let us shift the thread numbers (for the threads $\mathcal{T}_0, \ldots, \mathcal{T}_{P-1}$) so that $\mathcal{T}_0$ makes $\mathcal{T}_P$ fail on its first attempt. We distinguish two cases, depending on whether $G^{(P)}_0 > R^0_P - S_0$ or not.

We assume that $G^{(P)}_0 > R^0_P - S_0$. We know that $n \mapsto G^{(n)}_n$ is increasing on $[\![0, P-1]\!]$ and that $G^{(0)}_0 = 0$, hence let $n_0 = \min\{n \in [\![0, P-1]\!] \ ; \ G^{(n)}_n > R^0_P - S_0\}$. For all $k \in [\![0, n_0]\!]$, we have $R^k_P - S_k = k + R^0_P - (G^{(k)}_k + S_0 + k) = R^0_P - S_0 - G^{(k)}_k$ hence $R^k_P - S_k > 0$ and $R^k_P - S_k < R^0_P - S_0 < 1$. This shows that $\mathcal{T}_0, \ldots, \mathcal{T}_{n_0}$, because of their successes at $S_0, \ldots, S_{n_0}$, successively make $0^{\text{th}}, \ldots, n_0^{\text{th}}$ retries (respectively) of $\mathcal{T}_P$ fail. The next attempt for $\mathcal{T}_P$ is at $R^{n_0+1}_P$, which fulfills the following inequality: $R^{n_0+1}_P - (S_{n_0} + 1) <$

$S_{n_0+1} - (S_{n_0} + 1)$ since

$$R_P^{n_0+1} - S_{n_0+1} = (n_0 + 1 + R_P^0) - (G_{n_0+1}^{(n_0+1)} + S_0 + n_0 + 1)$$
$$R_P^{n_0+1} - S_{n_0+1} < 0.$$

$\mathcal{T}_{n_0+1}$ should have been the successful thread, but $\mathcal{T}_P$ starts a retry before S_{n_0+1}, and is therefore succeeding.

We consider now the reverse case by assuming that $G_0^{(P)} < R_P^0 - S_0$. With the previous line of reasoning, we can show that $\mathcal{T}_0, \ldots, \mathcal{T}_{P-1}$, because of their successes at $S_0, \ldots, S_{P-1}$, successively make $0^{\text{th}}, \ldots, (P-1)^{\text{th}}$ retries (respectively) of $\mathcal{T}_P$ fail. Then we are back in the same situation when $\mathcal{T}_0$ made $\mathcal{T}_P$ fail for the first time ($\mathcal{T}_0$ makes $\mathcal{T}_P$ fail), except that the success of $\mathcal{T}_0$ starts at $S_0' = S_0 + G_0^{(P)}$. As $G_0^{(P)} = q + r + f - P > 0$ and q, f and P are integers, we have that $G_0^{(P)} \geq r$. By the way, if we had $G_0^{(P)} > r$, we would have $G_0^{(P)} \geq 1 + r > R_P^0 - S_0$, which is absurd. S_0 makes indeed R_P^0 fail, therefore $G_0^{(P)}$ should be less than 1. Consequently, we are ensured that $G_0^{(P)} = r$. We define

$$k_0 = \left\lfloor \frac{R_P^0 - S_0}{r} \right\rfloor;$$

also, for every $k \in [\![1, k_0]\!]$, $r < R_P^0 - (S_0 + k \times r)$ and $r > R_P^0 - (S_0 + (k_0 + 1) \times r)$: the cycle of successes of $\mathcal{T}_0, \ldots, \mathcal{T}_{P-1}$ is executed k_0 times. Then the situation is similar to the first case, and $\mathcal{T}_P$ will succeed.

□

Lemma 2. *Let $\mathcal{S}$ be a weakly-formed seed, and $f = f\left((\mathcal{T}_i, S_i)_{i \in [\![0, P-2]\!]}\right)$. If, for all $n \in [\![0, P-1]\!]$, $G_n^{(f+1)} < 1$, then there exists later in the execution a well-formed seed $\mathcal{S}'$ for P threads such that $f(\mathcal{S}') = f + 1$.*

Proof. The proof is straightforward; $\mathcal{S}$ is actually a well-formed seed such that $f(\mathcal{S}) = f + 1$. Since $R_0^f - S_{P-1} < G_0^{(1)} < 1$, the first success of $\mathcal{T}_0$ after the success of $\mathcal{T}_{P-1}$ is its $f + 1^{\text{th}}$ retry. □

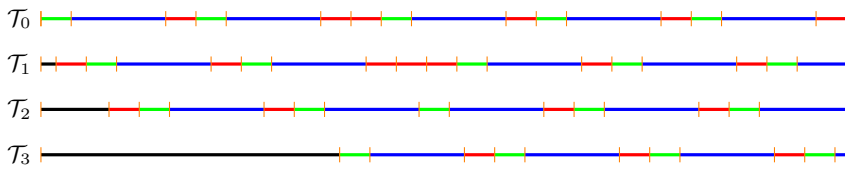


Figure 2.7: Lemma 3 configuration

Lemma 3. *Let $\mathcal{S}$ be a weakly-formed seed, and $f = f\left((\mathcal{T}_i, S_i)_{i\in[\![0,P-2]\!]}\right)$. If $G_f^{(f+1)} > 1$, and if the second success of $\mathcal{T}_{P-1}$ does not occur before the second success of $\mathcal{T}_{f-1}$, then we can find in the execution a well-formed seed $\mathcal{S}'$ for P threads such that $f(\mathcal{S}') = f$.*

Proof. Let us first remark that, by the definition of a weakly-formed seed, all threads will succeed once, in order. Then two ordered groups of threads will compete for each of the next successes, until $\mathcal{T}_{f-1}$ succeeds for the second time.

Let e be the smallest integer of $[\![f, P-1]\!]$ such that the second success of $\mathcal{T}_e$ occurs after the second success of $\mathcal{T}_{f-1}$. Let then $\mathcal{S}_1$ and $\mathcal{S}_2$ be the two groups of threads that are in competition, defined by

$$\mathcal{S}_1 = \{\mathcal{T}_n \ ; \ n \in [\![0, f-1]\!]\}$$
$$\mathcal{S}_2 = \{\mathcal{T}_n \ ; \ n \in [\![f, e-1]\!]\}$$

For all $n \in [\![0, e-1]\!]$, we note

$$rank(n) = \begin{cases} G_n^{(n+1)} & \text{if } \mathcal{T}_n \in \mathcal{S}_1 \\ G_n^{(n+1)} - 1 & \text{if } \mathcal{T}_n \in \mathcal{S}_2 \end{cases}.$$

We define σ, a permutation of $[\![0, e-1]\!]$ that describes the reordering of the threads during the round of the second successes, such that, for all $(i,j) \in [\![0, e-1]\!]^2$, $\sigma(i) < \sigma(j)$ if and only if $rank(i) < rank(j)$.

We also define a function that will help in expressing the $\sigma^{-1}(k)$'s:

$$\begin{array}{rccc} m_2: & [\![0, e-1]\!] & \longrightarrow & [\![f, e-1]\!] \\ & k & \longmapsto & \max\{\ell \in [\![f, e-1]\!] \ ; \ \mathcal{T}_\ell \in \mathcal{S}_2 \ ; \ \sigma(\ell) \leq k\} \end{array}.$$

We note that $rank\big|_{[\![0,f-1]\!]}$ is increasing, as well as $rank\big|_{[\![f,e-1]\!]}$. This shows that $\#\{\mathcal{T}_\ell \in \mathcal{S}_2 \ ; \ \sigma(\ell) \leq k\} = m_2(k) - (f-1)$. Consequently, if $\mathcal{T}_{\sigma^{-1}(k)} \in \mathcal{S}_2$, then

$$\begin{aligned} m_2(k) &= \#\{\mathcal{T}_\ell \in \mathcal{S}_2 \ ; \ \sigma(\ell) \leq k\} + f - 1 \\ &= \#\{\mathcal{T}_\ell \in \mathcal{S}_2 \ ; \ \ell \leq \sigma^{-1}(k)\} + f - 1 \\ &= \sigma^{-1}(k) - f + 1 + f - 1 \\ m_2(k) &= \sigma^{-1}(k). \end{aligned}$$

Conversely, if $\mathcal{T}_{\sigma^{-1}(k)} \in \mathcal{S}_1$, among $\{\mathcal{T}_{\sigma(n)} \ ; \ n \in [\![0, k]\!]\}$, there are exactly $m_2(k) - f + 1$ threads in $\mathcal{S}_2$, hence

$$\sigma^{-1}(k) = k + 1 - (m_2(k) - f + 1) - 1 = f + k - m_2(k) - 1.$$

In both cases, among $\{\mathcal{T}_{\sigma(n)} \ ; \ n \in [\![0, k]\!]\}$, there are exactly $m_2(k) - f + 1$ threads in $\mathcal{S}_2$, and $m_1(k) = k - (m_2(k) - f)$ threads in $\mathcal{S}_1$.

We prove by induction that after this first round, the next successes will be, respectively, achieved by $\mathcal{T}_{\sigma^{-1}(0)}$, $\mathcal{T}_{\sigma^{-1}(1)}$, ..., $\mathcal{T}_{\sigma^{-1}(e-1)}$. In the following, by "$k^{\text{th}}$ success", we mean k^{th} success after the first success of $\mathcal{T}_{P-1}$, starting from 0, and the R_i^j's denote the attempts of the second round.

Let $(\mathcal{P}_K)$ be the following property: for all $k \leq K$, the k^{th} success is achieved by $\mathcal{T}_{\sigma^{-1}(k)}$ at $R_{\sigma^{-1}(k)}^{f+k-\sigma^{-1}(k)}$. We assume $(\mathcal{P}_K)$ true, and we show that the $(K+1)^{\text{th}}$ success is achieved by $\mathcal{T}_{\sigma^{-1}(K+1)}$ at $R_{\sigma^{-1}(K+1)}^{f+K+1-\sigma^{-1}(K+1)}$.

We first show that if $\mathcal{T}_{\sigma^{-1}(K)} \in \mathcal{S}_1$, then

$$R_{m_2(K)+1}^{m_1(K)-1} > R_{\sigma^{-1}(K)}^{f+K-\sigma^{-1}(K)} > R_{m_2(K)}^{m_1(K)}. \tag{2.6}$$

On the one hand,

$$\begin{aligned} R_{\sigma^{-1}(K)}^{f+K-\sigma^{-1}(K)} &= K - \sigma^{-1}(K) + R_{\sigma^{-1}(K)}^{f} \\ &= K - \sigma^{-1}(K) + R_0^f + \sigma^{-1}(K) + G_{\sigma^{-1}(K)}^{(\sigma^{-1}(K))} \\ &= K + S_{P-1} + 1 + G_0^{(1)} + G_{\sigma^{-1}(K)}^{(\sigma^{-1}(K))} \\ R_{\sigma^{-1}(K)}^{f+K-\sigma^{-1}(K)} &= K + S_{P-1} + 1 + G_{\sigma^{-1}(K)}^{(\sigma^{-1}(K)+1)}. \end{aligned}$$

On the other hand,

$$\begin{aligned} R_{m_2(K)}^{f+K-m_2(K)} &= (m_2(K) - f) + R_f^{K-(m_2(K)-f)} + G_{m_2(K)}^{(m_2(K)-f)} \\ &= (m_2(K) - f) + K - (m_2(K) - f) + R_f^0 \\ &\quad + G_{m_2(K)}^{(m_2(K)-f)} \\ &= (m_2(K) - f) + K - (m_2(K) - f) + S_{P-1} \\ &\quad + 1 + (G_f^{(f+1)} - 1) + G_{m_2(K)}^{(m_2(K)-f)} \\ R_{m_2(K)}^{f+K-m_2(K)} &= K + S_{P-1} + 1 + G_{m_2(K)}^{(m_2(K)+1)} - 1. \end{aligned}$$

Therefore,

$$\begin{aligned} R_{\sigma^{-1}(K)}^{f+K-\sigma^{-1}(K)} - R_{m_2(K)}^{m_1(K)} &= R_{\sigma^{-1}(K)}^{f+K-\sigma^{-1}(K)} - R_{m_2(K)}^{f+K-m_2(K)} \\ &= G_{\sigma^{-1}(K)}^{(\sigma^{-1}(K)+1)} - \left(G_{m_2(K)}^{(m_2(K)+1)} - 1\right) \\ R_{\sigma^{-1}(K)}^{f+K-\sigma^{-1}(K)} - R_{m_2(K)}^{m_1(K)} &= rank\left(\sigma^{-1}(K)\right) - rank\left(m_2(K)\right). \end{aligned}$$

In a similar way, we can obtain that if $\mathcal{T}_{\sigma^{-1}(K)} \in \mathcal{S}_2$, then

$$R_{m_1(K)}^{m_2(K)} > R_{\sigma^{-1}(K)}^{f+K-\sigma^{-1}(K)} > R_{m_1(K)-1}^{m_2(K)+1}. \tag{2.7}$$

In addition, we recall that if $\mathcal{T}_{\sigma^{-1}(K)} \in \mathcal{S}_2$, $\sigma^{-1}(K) = m_2(K)$, thus the second inequality of Equation 2.6 becomes an equality, and if $\mathcal{T}_{\sigma^{-1}(K)} \in \mathcal{S}_1$, $\sigma^{-1}(K) = f + K - m_2(K) - 1$, hence the second inequality of Equation 2.7 becomes an equality.

Now let us look at which attempt of other threads $\mathcal{T}_{\sigma^{-1}(K)}$ made fail. From now on, and until explicitly said otherwise, we assume that $\mathcal{T}_{\sigma^{-1}(K)} \in \mathcal{S}_1$. According to Equation 2.6, we have

$$\begin{array}{rcl} R^{m_1(K)-1}_{m_2(K)+1} > & R^{f+K-\sigma^{-1}(K)}_{\sigma^{-1}(K)} > & R^{m_1(K)}_{m_2(K)} \\ R^{m_1(K)-j}_{m_2(K)+j} - R^{m_1(K)-1}_{m_2(K)+1} < & R^{m_1(K)-j}_{m_2(K)+j} - R^{f+K-\sigma^{-1}(K)}_{\sigma^{-1}(K)} < & R^{m_1(K)-j}_{m_2(K)+j} - R^{m_1(K)}_{m_2(K)} \\ G^{(j-1)}_{m_2(K)+j} < & R^{m_1(K)-j}_{m_2(K)+j} - R^{f+K-\sigma^{-1}(K)}_{\sigma^{-1}(K)} < & G^{(j)}_{m_2(K)+j} \end{array}$$

This holds for every $j \in [\![1, m_1(K)]\!]$, implying $j \leq f$, since there could not be more than f threads in $\mathcal{S}_1$. Therefore, as by assumptions gaps of at most f^{th} order are between 0 and 1,

$$0 < R^{m_1(K)-j}_{m_2(K)+j} - R^{f+K-\sigma^{-1}(K)}_{\sigma^{-1}(K)} < 1;$$

showing that the success of $\mathcal{T}_{\sigma^{-1}(K)}$ makes thread $\mathcal{T}_{m_2(K)+j}$ fail on its attempt at $R^{m_1(K)-j}_{m_2(K)+j}$, for all $j \in [\![1, m_1(K)]\!]$.

Since $\mathcal{T}_{\sigma^{-1}(K)} \in \mathcal{S}_1$, $\sigma^{-1}(K) = m_1(K) - 1$. Also, for all $j \in [\![0, f - 1 - m_1(K)]\!]$,

$$\begin{aligned} R^{m_2(K)-j}_{m_1(K)+j} - R^{f+K-\sigma^{-1}(K)}_{\sigma^{-1}(K)} &= R^{m_2(K)-j}_{m_1(K)+j} - R^{m_2(K)+1}_{m_1(K)-1} \\ &= \left(R^{m_2(K)-j}_{m_1(K)-1} + (j+1) + G^{(j+1)}_{m_1(K)+j}\right) \\ &\quad - \left(R^{m_2(K)-j}_{m_1(K)-1} + (j+1)\right) \\ R^{m_2(K)-j}_{m_1(K)+j} - R^{f+K-\sigma^{-1}(K)}_{\sigma^{-1}(K)} &= G^{(j+1)}_{m_1(K)+j} \end{aligned}$$

As a result, $\mathcal{T}_{\sigma^{-1}(K)}$ makes $\mathcal{T}_{m_1(K)+j}$ fail on its attempt at $R^{m_2(K)-j}_{m_1(K)+j}$, for all $j \in [\![0, f - 1 - m_1(K)]\!]$, and the next attempt will occur at $R^{m_2(K)-j+1}_{m_1(K)+j}$.

Altogether, the next attempt after the end of the success of $\mathcal{T}_{\sigma^{-1}(K)}$ for $\mathcal{T}_{m_1(K)+j}$ is $R^{m_2(K)-j+1}_{m_1(K)+j}$, for $j \in [\![0, f - 1 - m_1(K)]\!]$, and for $\mathcal{T}_{m_2(K)+j}$ is $R^{m_1(K)-j+1}_{m_2(K)+j}$, for all $j \in [\![1, m_1(K)]\!]$.

Additionally, a thread will begin a new retry loop, the 0^{th} retry being at $R^0_{m_2(K)+m_1(K)+1} = R^0_{f+K+1}$. We note that $f + K + 1$ could be higher than $P - 1$, referring to a thread whose number is more than $P - 1$. Actually,

if $n > P-1$, R_n^j refers to the j^{th} retry of $\mathcal{T}_{rank(n-P+1)}$, after its first two successes.

The two heads, *i.e.* the two smallest indices, of $\mathcal{S}_1 \cap \sigma^{-1}\left(\llbracket K+1, e-1 \rrbracket\right)$ and $\mathcal{S}_2 \cap \sigma^{-1}\left(\llbracket K+1, e-1 \rrbracket\right)$ will then compete for being successful. Indeed, within $\mathcal{S}_1$, for $j \in \llbracket 0, f-1-m_1(K) \rrbracket$,

$$R_{m_1(K)+j}^{m_2(K)-j+1} - R_{m_1(K)}^{m_2(K)+1} = G_{m_1(K)+j}^{(j)} > 0,$$

thus if someone succeeds in $\mathcal{S}_1$, it will be $\mathcal{T}_{m_1(K)}$. In the same way, for all $j \in \llbracket 1, m_1(K)+1 \rrbracket$,

$$R_{m_2(K)+j}^{m_1(K)-j+1} - R_{m_2(K)+1}^{m_1(K)} = G_{m_2(K)+j}^{(j-1)} > 0,$$

meaning that if someone succeeds in $\mathcal{S}_2$, it will be $\mathcal{T}_{m_2(K)+1}$.

Let us compare now those two candidates:

$$\begin{aligned}
R_{m_1(K)}^{m_2(K)+1} - R_{m_2(K)+1}^{m_1(K)} &= m_2(K) + 1 - f + S_{P-1} + m_1(K) + G_{m_1(K)}^{(m_1(K)+1)} \\
&\quad - \left(m_1(K) + R_f^0 + m_2(K) + 1 - f + G_{m_2(K)+1}^{(m_2(K)+1-f)} \right) \\
&= S_{P-1} - 1 + G_{m_1(K)}^{(m_1(K)+1)} \\
&\quad - \left(S_{P-1} + G_f^{(f+1)} - 1 + G_{m_2(K)+1}^{(m_2(K)+1-f)} \right) \\
&= G_{m_1(K)}^{(m_1(K)+1)} - \left(G_{m_2(K)+1}^{(m_2(K)+2)} - 1 \right) \\
R_{m_1(K)}^{m_2(K)+1} - R_{m_2(K)+1}^{m_1(K)} &= rank\left(m_1(K)\right) - rank\left(m_2(K)+1\right).
\end{aligned}$$

By definition, $\sigma^{-1}(K+1)$ is either $m_1(K)$ or $m_2(K)+1$ and corresponds to the next successful thread. We can follow the same line of reasoning in the case where $\mathcal{T}_{\sigma^{-1}(K)} \in \mathcal{S}_2$ and prove in this way that $(\mathcal{P}_{K+1})$ is true.

$(\mathcal{P}_0)$ is true, and the property spreads until $(\mathcal{P}_{e-1})$, where all threads of $\mathcal{S}_1$ and $\mathcal{S}_2$ have been successful, in the order ruled by σ^{-1}, *i.e.* $\mathcal{T}_{\sigma^{-1}(0)}$, …, $\mathcal{T}_{\sigma^{-1}(e-1)}$. And before those successes the threads $\mathcal{T}_e$, …, $\mathcal{T}_{P-1}$ have been successful as well. The seed composed of those successes is a well-formed seed. Given a thread, the gap between this thread and the next one in the new order could indeed not be higher than the gap in the previous order with its next

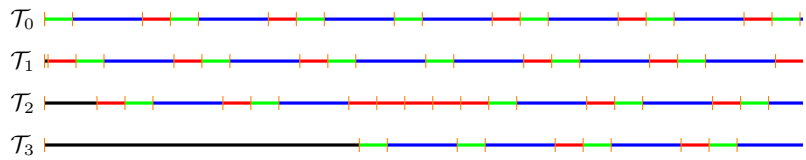


Figure 2.8: Lemma 4 configuration

thread. Also the f^{th} order gaps remain smaller than 1. And as $\mathcal{T}_e$ succeeds the second time after f failures, it means that the new seed $\mathcal{S}''$ is such that $f(\mathcal{S}'') = f$. □

Lemma 4. *Let $\mathcal{S}$ be a weakly-formed seed, and $f = f\left((\mathcal{T}_i, S_i)_{i\in[\![0,P-2]\!]}\right)$. If $G_f^{(f+1)} > 1$ and if the second success of $\mathcal{T}_{P-1}$ occurs before the second success of $\mathcal{T}_{f-1}$, then we can find in the execution a well-formed seed $\mathcal{S}'$ for P threads such that $f(\mathcal{S}') = f$.*

Proof. Until the second success of $\mathcal{T}_{P-1}$, the execution follows the same pattern as in Lemma 3. Actually, the case invoked in the current lemma could have been handled in the previous lemma, but it would have implied tricky notations, when we referred to $\mathcal{T}_{rank(n-P+1)}$. Let us deal with this case independently then, and come back to the instant where $\mathcal{T}_{P-1}$ succeeds for the second time.

We had $0 < R_{f-1}^0 - S_{P-1} = G_{f-1}^{(f)} < 1$. For the thread $\mathcal{T}_{\sigma(j)}$ to succeed at its k^{th} retry after the first success of $\mathcal{T}_{P-1}$ and before $\mathcal{T}_{f-1}$, it should necessary fill the following condition: $j+1 < R_{\sigma(j)}^k - S_{P-1} < j+1+G_{f-1}^{(f)}$. This holds also for the second success of $\mathcal{T}_{P-1}$, which implies that $P' < S_{P-1} + 1 + q + r + h - S_{P-1} < P' + G_{f-1}^{(f)}$, where h is the number of failures of $\mathcal{T}_{P-1}$ before its second success and P' is the number of successes between the two successes of $\mathcal{T}_{P-1}$. As $G_{f-1}^{(f)} < 1$, and q, P' and h are non-negative integers, we have $r < G_{f-1}^{(f)}$ and $h = P' - 1 - q$.

To conclude, as any gap at any order is less than the gap between the two successes of $\mathcal{T}_{P-1}$, which is $r < 1$, we found a well-formed seed for P' threads.

Finally any other thread will eventually succeed (see Lemma 1). We can renumber the threads such that $\mathcal{T}_{P'}$ is the first thread that is not in the well-formed seed to succeed, and the threads of the well-formed seed succeeded previously as $\mathcal{T}_0, \ldots, \mathcal{T}_{P'-1}$. As explained before, for all $(k,n) \in [\![0, P'-1]\!]^2$, $G_n^{(k)} < G_n^{(n)} = r$. With the new thread, the first order gaps are changed by decomposing $G_0^{(1)}$ into $G_{P'}^{(1)}$ and the new $G_0^{(1)}$. All gaps can only be decreased, hence we have a new well-formed seed for $P'+1$ threads. We repeat the process until all threads have been encountered, and obtain in the end $\mathcal{S}'$, a well-formed seed with P threads such that $f(\mathcal{S}') = P - 1 - q$, which is an optimal cyclic execution.

Still, as $\mathcal{T}_f$ succeeds between two successes of $\mathcal{T}_{P-1}$ that are separated by r, we had, in the initial configuration: $G_{P-1}^{(P-1-f)} < r$. As, in addition, we have both $G_{f-1}^{(f)} < 1$ and $G_f^{(1)} < 1$, we conclude that the lagging time was initially less than $2 + r$. By hypothesis, we know that $G_f^{(f+1)} > 1$, which implies that, before the entry of the new thread, the lagging time was $1 + r$. In the final execution with one more thread, the lagging time is r and we have one more success in the cycle, thus $f(\mathcal{S}') = f$. □

Theorem 2. *Assuming $r \neq 0$, if a new thread is added to an $(f, P-1)$-cyclic execution, then all the threads will eventually form either an (f, P)-cyclic execution, or an $(f+1, P)$-cyclic execution.*

Proof. According to Lemma 1, the new thread will eventually succeed. In addition, we recall that Properties 1 and 2 ensure that before the first success of the new thread, any set of $P-1$ consecutive successes is a well-formed seed with $P-1$ threads. We then consider a seed (we number the threads accordingly, and number the new thread as $\mathcal{T}_{P-1}$) such that the success of the new thread occurs between the success of $\mathcal{T}_{P-2}$ and $\mathcal{T}_0$; we obtain in this way a weakly-formed seed $\mathcal{S} = (\mathcal{T}_n, S_n)_{n \in [\![0, P-1]\!]\&}$. We differentiate between two cases.

Firstly, if for all $n \in [\![0, P-1]\!]$, $G_n^{(f+1)} < 1$, according to Lemma 2, we can find later in the execution a well-formed seed $\mathcal{S}'$ for P threads such that $f(\mathcal{S}') = f + 1$, hence we reach eventually an $(f+1, P)$-cyclic execution.

Let us assume now that this condition is not fulfilled. There exists $n_0 \in$

$[\![0, P-1]\!]$ such that $G_{n_0}^{(f+1)} > 1$. We shift the thread numbers, such that n_0 is now f, and we have then $G_f^{(f+1)} > 1$. Then two cases are feasible. If the second success of $\mathcal{T}_{P-1}$ occurs before the second success of $\mathcal{T}_{f-1}$, then Lemma 3 shows that we will reach an (f, P)-cyclic execution. Otherwise, from Lemma 3, we conclude that an (f, P)-cyclic execution will still occur. □

2.4.3 Throughput Bounds

Firstly we calculate the expression of throughput and the expected number of threads inside the retry loop (that is needed when we gather expansion and wasted retries). Then we exhibit upper and lower bounds on both throughput and the number of failures, and show that those bounds are reached. Finally, we give the worst case on the number of wasted retries.

Lemma 5. *In an (f, P)-cyclic execution, the throughput is*

$$T = \frac{P}{q + r + 1 + f}. \tag{2.8}$$

Proof. By definition, the execution is periodic, and the period lasts $q+r+1+f$ units of time. As P successes occur during this period, we end up with the claimed expression. □

Lemma 6. *In an (f, P)-cyclic execution, the average number of threads P_{rl} in the retry loop is given by*

$$P_{rl} = P \times \frac{f+1}{q + r + f + 1}.$$

Proof. Within a period, each thread spends $f + 1$ units of time in the retry loop, among the $q + r + f + 1$ units of time of the period, hence the Lemma. □

Lemma 7. *The number of failures is not less than $f^{(-)}$, where*

$$f^{(-)} = \begin{cases} P - q - 1 & \text{if } q \leq P - 1 \\ 0 & \text{otherwise} \end{cases}, \text{and,} \qquad T \leq \begin{cases} \frac{P}{P+r} & \text{if } q \leq P - 1 \\ \frac{P}{q+r+1} & \text{otherwise.} \end{cases} \tag{2.9}$$

Proof. According to Equation 2.8, the throughput is maximized when the number of failures is minimized. In addition, we have two lower bounds on the number of failures: (i) $f \geq 0$, and (ii) P successes should fit within a period, hence $q + 1 + f \geq P$. Therefore, if $P - 1 - q < 0$, $T \leq P/(q + r + 1 + 0)$, otherwise,

$$T \leq \frac{P}{q + r + 1 + P - 1 - q} = \frac{P}{P + r}.$$

□

Remark 3. We notice that if $q > P - 1$, the upper bound in Equation 2.9 is actually the same as the immediate upper bound described in Section 2.3.2.1. However, if $q \leq P - 1$, Equation 2.9 refines the immediate upper bound.

Lemma 8. *The number of failures is bounded by*

$$f \leq f^{(+)} = \left\lfloor \frac{1}{2} \left((P - 1 - q - r) + \sqrt{(P - 1 - q - r)^2 + 4P} \right) \right\rfloor,$$

and accordingly, the throughput is bounded by

$$T \geq \frac{P}{q + r + 1 + f^{(+)}}.$$

Proof. We show that a necessary condition so that an (f, P)-cyclic execution, whose lagging time is ℓ, exists, is $f \times (\ell + r) < P$. According to Property 1, any set of P consecutive successes is a well-formed seed with P threads. Let $\mathcal{S}$ be any of them. As we have f failures before success, Theorem 1 ensures that for all $n \in [\![0, P - 1]\!]$, $G_n^{(f)} < 1$. We recall that for all $n \in [\![0, P - 1]\!]$, we also have $G_n^{(P)} = \ell + r$.

On the one hand, we have

$$\begin{aligned} \sum_{n=0}^{P-1} G_n^{(f)} &= \sum_{n=0}^{P-1} \sum_{j=n-f+1}^{n} G_{j \bmod P}^{(1)} \\ &= f \times \sum_{n=0}^{P-1} G_n^{(1)} \\ \sum_{n=0}^{P-1} G_n^{(f)} &= f \times (\ell + r). \end{aligned}$$

On the other hand, $\sum_{n=0}^{P-1} G_n^{(f)} < \sum_{n=0}^{P-1} 1 = P$.

Altogether, the necessary condition states that $f \times (\ell + r) < P$, which can be rewritten as $f \times (q+1+f-P+r) < P$. The proof is complete since minimizing the throughput is equivalent to maximizing the number of failures. □

Lemma 9. *For each of the bounds defined in Lemmas 7 and 8, there exists an (f, P)-cyclic execution that reaches the bound.*

Proof. According to Lemmas 7 and 8, if an (f, P)-cyclic execution exists, then the number of failures is such that $f^{(-)} \leq f \leq f^{(+)}$. We show now that this double necessary condition is also sufficient. We consider f such that $f^{(-)} \leq f \leq f^{(+)}$, and build a well-formed seed $\mathcal{S} = (\mathcal{T}_i, S_i)_{i \in [\![0, P-1]\!]}$.

For all $n \in [\![0, P-1]\!]$, we define S_i as

$$S_n = n \times \left(\frac{q+1+f-P+r}{P} + 1 \right).$$

We first show that $f(\mathcal{S}) = f$.
By definition, $f(\mathcal{S}) = \max(0, \lceil S_{P-1} - S_0 - q - r \rceil)$, we have then;

$$\begin{aligned} f(\mathcal{S}) &= \max\left(0, \left\lceil (P-1) \times \left(\frac{q+1+f-P+r}{P} + 1 \right) - q - r \right\rceil \right) \\ &= \max\left(0, \left\lceil (P-1-q-r) + (q+1+f-P+r) - \frac{q+1+f-P+r}{P} \right\rceil \right) \\ f(\mathcal{S}) &= \max\left(0, \left\lceil f - \frac{q+1+f-P+r}{P} \right\rceil \right). \end{aligned}$$

Firstly, we know that $q+1+f-P \geq 0$, thus if $f = 0$, then the second term of the maximum is not positive, and $f(\mathcal{S}) = 0 = f$. Secondly, if $f > 0$, then according to Lemma 7, $(q+1+f-P+r)/P < 1/f \leq 1$. As we also have $(q+1+f-P+r)/P \geq 0$, we conclude that $f(\mathcal{S}) = \left\lceil f - \frac{q+1+f-P+r}{P} \right\rceil = f$.

Additionally, for all $n \in [\![0, P-1]\!]$,

$$
\begin{aligned}
G_n^{(f)} &= \begin{cases} S_n - S_{n-f} - f & \text{if } n > f \\ S_n - S_{P+n-f} + 1 + q + r & \text{otherwise} \end{cases} \\
&= \begin{cases} n \times \left(\frac{q+1+f-P+r}{P} + 1\right) \\ -(n-f) \times \left(\frac{q+1+f-P+r}{P} + 1\right) - f \\ \\ n \times \left(\frac{q+1+f-P+r}{P} + 1\right) \\ -(P+n-f) \times \left(\frac{q+1+f-P+r}{P} + 1\right) + 1 + q + r \end{cases} \\
&= \begin{cases} f \times \frac{q+1+f-P+r}{P} \\ \\ -(P-f) - (q+1+f-P+r) + \\ f \times \frac{q+1+f-P+r}{P} + 1 + q + r \end{cases} \\
G_n^{(f)} &= f \times \frac{w+r}{P}
\end{aligned}
$$

As $w \geq 0$ and $f \geq 0$, $G_n^{(f)} > 0$. Since $f \leq f^{(+)}$, $G_n^{(f)} < 1$. Theorem 1 implies that $\mathcal{S}$ is a well-formed seed that leads to an (f, P)-cyclic execution.

We have shown that for all f such that $f^{(-)} \leq f \leq f^{(+)}$ there exists an (f, P)-cyclic execution; in particular there exist an $(f^{(+)}, P)$-cyclic execution and an $(f^{(-)}, P)$-cyclic execution. □

Corollary 1. *The highest possible number of wasted repetitions is* $\left\lceil \sqrt{P} - 1 \right\rceil$ *and is achieved when* $P = q + 1$.

Proof. The highest possible number of wasted repetitions $\tilde{w}(P)$ with P threads is given by

$$\tilde{w}(P) = f^{(+)} - f^{(-)} = \left\lfloor \frac{1}{2}\left(-a(P) + \sqrt{a(P)^2 + 4P}\right) - f^{(-)} \right\rfloor.$$

Let a and h be the functions respectively defined as $a(P) = q + 1 - P + r$, which implies $a'(P) = -1$, and $h(P) = (-a(P) + \sqrt{a(P)^2 + 4P})/2 - f^{(-)}$, so that $\tilde{w}(P) = \lfloor h(P) \rfloor$.

Let us first assume that $a(P) > 0$. In this case, $q \geq P - 1$, hence $f^{(-)} = 0$. We have

$$2h'(P) = 1 + \frac{-2a(P) + 4}{2\sqrt{a(P)^2 + 4P}}$$
$$2h'(P) = 2 \times \frac{2 - a(P) + \sqrt{a(P)^2 + 4P}}{2\sqrt{a(P)^2 + 4P}}$$

Therefore, $h'(P)$ is negative if and only if $\sqrt{a(P)^2 + 4P} < a(P) - 2$. It cannot be true if $a(P) < 2$. If $a(P) \geq 2$, then the previous inequality is equivalent to $a(P)^2 + 4P < a(P)^2 - 4a(P) + 4$, which can be rewritten in $q + 1 + r < 1$, which is absurd. We have shown that h is increasing in $]0, q + 1]$.

Let us now assume that $a(P) \leq 0$. In this case, $q < P - 1$, hence $f^{(-)} = P - q - 1$, and $h(P) = \left(a(P) + \sqrt{a(P)^2 + 4P}\right)/2 - r$. Assuming $h'(P)$ to be positive leads to the same absurd inequality $q + 1 + r < 1$, which proves that h is decreasing on $[q + 2, +\infty[$.

Also, the maximum number of wasted repetitions is achieved as $P = q + 1$ or $P = q + 2$. Since

$$h(q+1) = \frac{1}{2}\left(-r + \sqrt{r^2 + 4P}\right) > \frac{1}{2}\left(-(r+1) + \sqrt{r^2 + 4P}\right) = h(q+2),$$

the maximum number of wasted repetitions is $\tilde{w}(q + 1)$. In addition,

$$\begin{array}{rcccl} \frac{1}{2}\left(-r + \sqrt{4P}\right) & < & h(q+1) & < & \frac{1}{2}\left(-r + \sqrt{r^2} + \sqrt{4P}\right) \\ \sqrt{P} - \frac{r}{2} & < & h(q+1) & < & \sqrt{P} \\ \sqrt{P} - 1 & \leq & h(q+1) & < & \sqrt{P} \end{array}$$

We conclude that the maximum number of wasted repetitions is $\left\lceil \sqrt{P} - 1 \right\rceil$. $\square$

2.5 Expansion and Complete Throughput Estimation

2.5.1 Expansion

Interference of threads does not only lead to logical conflicts but also to hardware conflicts which impact the performance significantly. We model the behavior of the cache coherency protocols which determine the interaction of overlapping *Read*s and *CAS*s. By taking MESIF [14] as basis, we come up with the following assumptions. When executing an atomic *CAS*, the core gets the cache line in exclusive state and does not forward it to any other requesting core until the instruction is retired. Therefore, requests stall for the release of the cache line which implies serialization. On the other hand, ongoing *Read*s can overlap with other operations. As a result, a *CAS* introduces expansion only to overlapping *Read* and *CAS* operations that start after it, as illustrated in Figure 2.4. As a remark, we ignore memory bandwidth issues which are negligible for our study.

Furthermore, we assume that *Read*s that are executed just after a *CAS* do not experience expansion (as the thread already owns of the data), which takes effect at the beginning of a retry following a failing attempt. Thus, read expansions need only to be considered before the 0^{th} retry. In this sense, read expansion can be moved to parallel section and calculated in the same way as *CAS* expansion is calculated.

To estimate expansion, we consider the delay that a thread can introduce, provided that there is already a given number of threads in the retry loop. The starting point of each *CAS* is a random variable which is distributed uniformly within an expanded retry. The cost function d provides the amount of delay that the additional thread introduces, depending on the point where the starting point of its *CAS* hits. By using this cost function we can formulate the expansion increase that each new thread introduces and derive the differential equation below to calculate the expansion of a *CAS*.

Lemma 10. *The expansion of a CAS operation is the solution of the following*

system of equations:

$$\begin{cases} e'\left(P_{rl}\right) & = & cc \times \dfrac{\frac{cc}{2} + e\left(P_{rl}\right)}{rc + cw + cc + e\left(P_{rl}\right)} \\ e\left(P_{rl}^{(0)}\right) & = & 0 \end{cases} , \quad \begin{array}{l} \textit{where } P_{rl}^{(0)} \textit{ is the} \\ \textit{point where} \\ \textit{expansion begins.} \end{array}$$

Proof. We compute $e\left(P_{rl}+h\right)$, where $h \leq 1$, by assuming that there are already P_{rl} threads in the retry loop, and that a new thread attempts to *CAS* during the retry, within a probability h.

$$\begin{aligned} e\left(P_{rl}+h\right) &= e\left(P_{rl}\right) + h \times \int_0^{rlw^{(+)}} \frac{d\left(t\right)}{rlw^{(+)}}\, dt \\ &= e\left(P_{rl}\right) + h \times \Big(\int_0^{rc+cw-cc} \frac{d\left(t\right)}{rlw^{(+)}}\, dt \\ &\qquad + \int_{rc+cw-cc}^{rc+cw} \frac{d\left(t\right)}{rlw^{(+)}}\, dt \\ &\qquad + \int_{rc+cw}^{rc+cw+e(P_{rl})} \frac{d\left(t\right)}{rlw^{(+)}}\, dt \\ &\qquad + \int_{rc+cw+e(P_{rl})}^{rlw^{(+)}} \frac{d\left(t\right)}{rlw^{(+)}}\, dt \Big) \\ &= e\left(P_{rl}\right) + h \times \Big(\int_{rc+cw-cc}^{rc+cw} \frac{t}{rlw^{(+)}}\, dt \\ &\qquad + \int_{rc+cw}^{rc+cw+e(P_{rl})} \frac{cc}{rlw^{(+)}}\, dt \Big) \\ e\left(P_{rl}+h\right) &= e\left(P_{rl}\right) + h \times \frac{\frac{cc^2}{2} + e\left(P_{rl}\right) \times cc}{rlw^{(+)}} \end{aligned}$$

This leads to $\dfrac{e\left(P_{rl}+h\right) - e\left(P_{rl}\right)}{h} = \dfrac{\frac{cc^2}{2} + e\left(P_{rl}\right) \times cc}{rlw^{(+)}}$. When making h tend to 0, we finally obtain

$$e'\left(P_{rl}\right) = cc \times \frac{\frac{cc}{2} + e\left(P_{rl}\right)}{rc + cw + cc + e\left(P_{rl}\right)}.$$

□

2.5.2 Throughput Estimate

It remains to combine hardware and logical conflicts in order to obtain the final upper and lower bounds on throughput. We are given as an input the expected number of threads P_{rl} inside the retry loop. We firstly compute the expansion accordingly, by solving numerically the differential equation of Lemma 17. As explained in the previous subsection, we have $pw^{(+)} = pw + e$, and $rlw^{(+)} = rc + cw + e + cc$. We can then compute q and r, that is the input set (together with the total number of threads P) of the method described in Section 2.4. Assuming that the initialization times of the threads are spaced enough, the execution will superimpose an (f, P)-cyclic execution. Thanks to Lemma 6, we can compute the average number of threads inside the retry loop, that we note by $h_f(P_{rl})$. A posteriori, the solution is consistent if this average number of threads inside the retry loop $h_f(P_{rl})$ is equal to the expected number of threads P_{rl} that has been given as an input.

Several (f, P)-cyclic executions belong to the domain of the possible outcomes, but we are interested in upper and lower bounds on the number of failures f. We can compute them through Lemmas 7 and 8, along with their corresponding throughput and average number of threads inside the retry loop. We note by $h^{(+)}(P_{rl})$ and $h^{(-)}(P_{rl})$ the average number of threads for the lowest number of failures and highest one, respectively. Our aim is finally to find $P_{rl}^{(-)}$ and $P_{rl}^{(+)}$, such that $h^{(+)}(P_{rl}^{(+)}) = P_{rl}^{(+)}$ and $h^{(-)}(P_{rl}^{(-)}) = P_{rl}^{(-)}$. If several solutions exist, then we want to keep the smallest, since the retry loop stops to expand when a stable state is reached.

Note that we also need to provide the point where the expansion begins. It begins when we start to have failures, while reducing the parallel section. Thus this point is $2(P-1)rlw^{(-)}$ (resp. $(P-1)rlw^{(-)}$) for the lower (resp. upper) bound on the throughput.

Theorem 3. *Let (x_n) be the sequence defined recursively by $x_0 = 0$ and $x_{n+1} = h^{(+)}(x_n)$. If $pw \geq rc + cw + cc$, then*

$$P_{rl}^{(+)} = \lim_{n \to +\infty} x_n.$$

Proof. First of all, the average number of threads belongs to $]0, P[$, thus for all $x \in [0, P]$, $0 < h^{(+)}(x) < P$. In particular, we have $h^{(+)}(0) > 0$, and $h^{(+)}(P) < P$, which proves that there exist one fixed point for $h^{(+)}$.

In addition, we show that $h^{(+)}$ is a non-decreasing function. According to Lemma 6,

$$h^{(+)}(P_{rl}) = P \times \frac{1 + f^{(-)}}{q + r + f^{(-)} + 1},$$

where all variables except P depend actually on P_{rl}. We have

$$q = \left\lfloor \frac{pw + e}{rlw^{(-)} + e} \right\rfloor \text{ and } r = \frac{pw + e}{rlw^{(-)} + e} - q,$$

hence, if $pw \geq rlw^{(-)}$, q and r are non-increasing as e is non-decreasing, which is non-decreasing with P_{rl}. Since $f^{(-)}$ is non-decreasing as a function of q, we have shown that if $pw \geq rlw^{(-)}$, $h^{(+)}$ is a non-decreasing function.

Finally, the proof is completed by the theorem of Knaster-Tarski. □

The same line of reasoning holds for $h^{(-)}$ as well. As a remark, w point out that when $pw < rlw^{(-)}$, we scan the interval of solution, and have no guarantees about the fact that the solution is the smallest one; still this corresponds to very extreme cases.

2.5.3 Several Retry Loops

We consider here a lock-free algorithm that, instead of being a loop over one parallel section and one retry loop, is composed of a loop over a sequence of alternating parallel sections and retry loops. We show that this algorithm is equivalent to an algorithm with only one parallel section and one retry loop, by proving the intuition that the longest retry loop is the only one that fails and hence expands.

2.5.3.1 Problem Formulation

In this subsection, we consider an execution such that each spawned thread runs Procedure Combined in Figure 2.9. Each thread executes a linear combination

of S independent retry loops, *i.e.* operating on separate variables, interleaved with parallel sections. We note now as $rlw_i^{(+)}$ and $pw_i^{(+)}$ the size of a retry of the i^{th} retry loop and the size of the i^{th} parallel section, respectively, for each $i \in [\![1, S]\!]$. As previously, q_i and r_i are defined such that $pw_i^{(+)} = (q_i + r_i) \times rlw_i^{(+)}$, where q_i is a non-negative integer and r_i is smaller than 1.

The Procedure Combined executes the retry loops and parallel sections in a cyclic fashion, so we can normalize the writing of this procedure by assuming that a retry of the 1^{st} retry loop is the longest one. More precisely, we consider the initial algorithm, and we define i_0 as

$$i_0 = \min \operatorname{argmax}_{i \in [\![1,S]\!]} rlw_i^{(+)}.$$

We then renumber the retry loops such that the new ordering is $i_0, \ldots, S, 1, \ldots, i_0 - 1$, and we add in Initialization the first parallel sections and retry loops on access points from 1 to i_0 — according to the initial ordering.

One success at the system level is defined as one success of the last *CAS*, and the throughput is defined accordingly. We note that in steady-state, all retry loops have the same throughput, so the throughput can be computed from the throughput of the 1^{st} retry loop instead.

Procedure Combined

```
1 Initialization();
2 while ! done do
3     for i ← 1 to S do
4         Parallel_Work(i);
5         while ! success do
6             current ← Read(AP[i]);
7             new ← Critical_Work(i,current);
8             success ← CAS(AP, current, new);
```

Figure 2.9: Thread procedure with several retry loops

2.5.3.2 Wasted Retries

Lemma 11. *Unsuccessful retry loops can only occur in the* 1^{st} *retry loop.*

Proof. We note $(t_n)_{n\in[1,+\infty[}$ the sequence of the thread numbers that succeeds in the 1^{st} retry loop, and $(s_n)_{n\in[1,+\infty[}$ the sequence of the corresponding time where they exit the retry loop. We notice that by construction, for all $n \in [1,+\infty[$, $s_n < s_{n+1}$. Let, for $i \in [\![2, S]\!]$ and $n \in [1,+\infty[$, $(\mathcal{P}_{i,n})$ be the following property: for all $i' \in [\![2, i]\!]$, and for all $n' \in [\![1, n]\!]$, the thread $\mathcal{T}_{t_{n'}}$ succeeds in the i^{th} retry loop at its first attempt.

We assume that for a given (i, n), $(\mathcal{P}_{i+1,n})$ and $(\mathcal{P}_{i,n+1})$ is true, and show that $(\mathcal{P}_{i+1,n+1})$ is true. As the threads $\mathcal{T}_{t_n}$ and $\mathcal{T}_{t_{n+1}}$ do not have any failure in the first i retry loops, their entrance time in the $i + 1^{\text{th}}$ retry loop is given by

$$s_n + \sum_{i'=1}^{i} (rlw_{i'}^{(+)} + pw_{i'}^{(+)}) + pw_{i+1}^{(+)} = X_1 \text{ and}$$

$$s_{n+1} + \sum_{i'=1}^{i} (rlw_{i'}^{(+)} + pw_{i'}^{(+)}) + pw_{i+1}^{(+)} = X_2,$$

respectively. Thread $\mathcal{T}_{t_n}$ does not fail in the $i + 1^{\text{th}}$ retry loop, hence exits at

$$X_1 + rlw_{i+1}^{(+)} < X_1 + rlw_1^{(+)} < X_2.$$

As the previous threads $\mathcal{T}_{n-1}, \ldots, \mathcal{T}_1$ exits the i^{th} retry loop before $\mathcal{T}_n$, and next threads $\mathcal{T}_{n'}$, where $n' > n + 1$, enters this retry loop after $\mathcal{T}_{n+1}$, this implies that the thread $\mathcal{T}_{t_{n+1}}$ succeeds in the $i + 1^{\text{th}}$ retry loop at its first attempt, and $(\mathcal{P}_{i+1,n+1})$ is true.

Regarding the first thread that succeeds in the first retry loop, we know that he successes in any retry loop since there is no other thread to compete with. Therefore, for all $i \in [\![2, S]\!]$, $(\mathcal{P}_{i,1})$ is true. Then we show by induction that all $(\mathcal{P}_{2,n})$ is true, then all $(\mathcal{P}_{3,n})$, *etc.*, until all $(\mathcal{P}_{S,n})$, which concludes the proof. □

Theorem 4. *The multi-retry loop Procedure Combined is equivalent to the Procedure AbstractAlgorithm, where*

$$pw^{(+)} = pw_1^{(+)} + \sum_{i=2}^{S} \left(pw_i^{(+)} + rlw_i^{(+)} \right) \quad \textit{and} \quad rlw^{(+)} = rlw_1^{(+)}.$$

Proof. According to Lemma 11 there is no failure in other retry loop than the first one; therefore, all retry loops have a constant duration, and can thus be considered as parallel sections. □

2.5.3.3 Expansion

The expansion in the retry loop starts as threads fail inside this retry loop. When threads are launched, there is no expansion, and Lemma 11 implies that if threads fail, it should be inside the first retry loop, because it is the longest one. As a result, there will be some stall time in the memory accesses of this first retry loop, *i.e.* expansion, and it will get even longer. Failures will thus still occur in the first retry loop: there is a positive feedback on the expansion of the first retry loop that keeps this first retry loop as the longest one among all retry loops. Therefore, in accordance to Theorem 4, we can compute the expansion by considering the equivalent single-retry loop procedure described in the theorem.

2.6 Experimental Evaluation

We validate our model and analysis framework through successive steps, from synthetic tests, capturing a wide range of possible abstract algorithmic designs, to several reference implementations of extensively studied lock-free data structure designs that include cases with non-constant parallel work and critical work.

2.6.1 Setting

We have conducted experiments on an Intel ccNUMA workstation system. The system is composed of two sockets, that is equipped with Intel Xeon E5-2687W v2 CPUs with frequency band 1.2-3.4. GHz The physical cores have private L1, L2 caches and they share an L3 cache, which is $25\,\mathrm{MB}$. In a socket, the ring interconnect provides L3 cache accesses and core-to-core communication. Due to the bi-directionality of the ring interconnect, uncontended latencies for intra-socket communication between cores do not show significant variability.

Our model assumes uniformity in the *CAS* and *Read* latencies on the shared cache line. Thus, threads are pinned to a single socket to minimize non-uniformity in *Read* and *CAS* latencies. In the experiments, we vary the number of threads between 4 and 8 since the maximum number of threads that can be used in the experiments are bounded by the number of physical cores that reside in one socket.

As mentioned before, the latencies of *CAS* and *Read* are parameters of our model. We used the methodology described in [15] to measure latencies of these operations in a benchmark program by using two threads that are pinned to the same socket. The aim is to bring the cache line into the state used in our model. Our assumption is that the *Read* is conducted on an invalid line. For *CAS*, the state of the cache line is assumed to be forward, shared or invalid. For any of these states of the cache line, *CAS* requests it for exclusive ownership, that compels invalidation in other cores, which in turn incurs a two-way communication. Thus, the latency of *CAS* does not show negligible variability with respect to the possible states of the cache line that we have assumed, as also revealed in our latency benchmarks.

As for the computation cost, the work inside the parallel section (whose latency is denoted by pw) is implemented by a dummy for-loop of *Pause* instructions. For synthetic tests, the critical work inside the retry loop (whose latency is denoted by cw) is also implemented in the same way.

In all figures, y-axis provides the throughput, which is the number of successful operations completed per millisecond. Parallel work latency (pw) is represented on x-axis in cycles. The critical work latency (cw) is given at the

top of the graphs. Number of threads that execute the algorithm is often given at the top of the graphs or otherwise at the captions. The graphs contain the high and low estimates that we derive in this work (see Section 2.4), corresponding to the lower and upper bound on the wasted retries respectively, and an additional curve that shows the average of them. They are referred to as *"Model Low"*, *"Model High"* and *"Model Average"* in the figures. Also, figures include a curve that provides the results of real measurements that are conducted on the system that is mentioned before, and this curve is referred to as *"Real Measurements"*.

2.6.2 Synthetic Tests

2.6.2.1 Single retry loop

For the evaluation of our model, we first create synthetic tests that emulate different design patterns of lock-free data structures (value of cw) and different application contexts (value of pw). As described in the previous subsection, in the Procedure AbstractAlgorithm, the amount of work in both the parallel section and the retry loop are implemented as dummy loops, whose costs are adjusted through the number of iterations in the loop.

Generally speaking, in Figure 2.10, we observe two main behaviors: when pw is high, the data structure is not contended, and threads can operate without failure (unsuccessful retries). When pw is low, the data structure is contended, and depending on the size of cw (that drives the expansion) a steep decrease in throughput or just a roughly constant bound on the performance is observed.

The position of the experimental curve between the high and low estimates, depends on cw. It can be observed that the experimental curve mostly tends upwards as cw gets smaller, possibly because the serialization of the *CAS*s helps the synchronization of the threads.

Another interesting fact is the waves appearing on the experimental curve, especially when the number of threads is low or the critical work big. This behavior is originating because of the variation of r with the change of parallel work, a fact that is captured by our analysis.

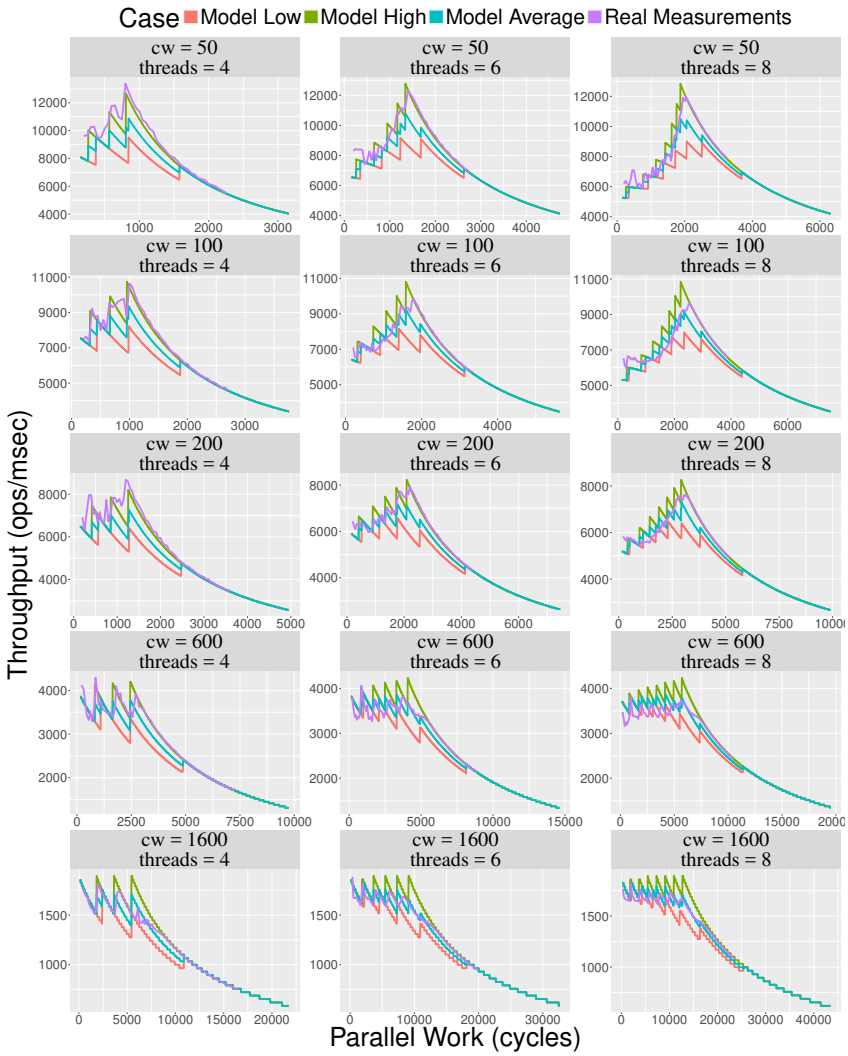


Figure 2.10: Synthetic program

2.6.2.2 Several retry loops

We have created experiments by combining several retry loops (See section 2.5.3), each operating on an independent variable which is aligned to a cache line. In

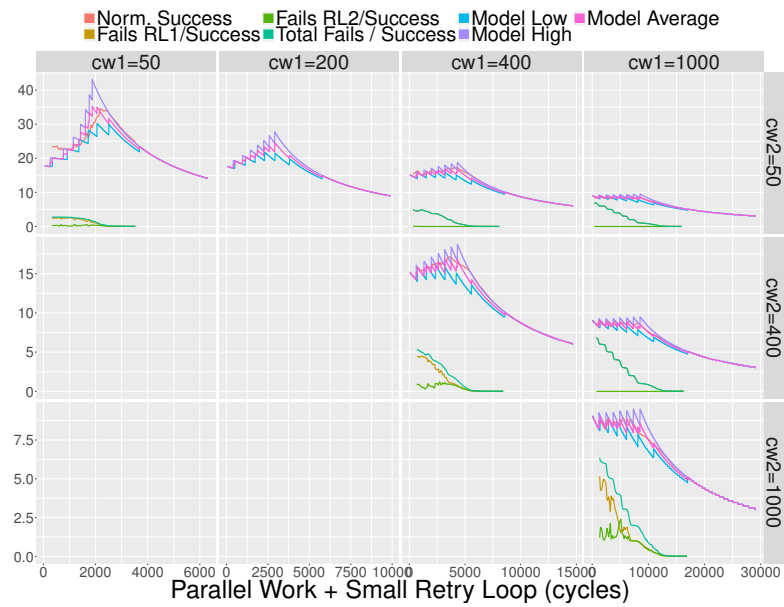


Figure 2.11: Multiple retry loops with 8 threads

Figure 2.11, we provide results for synthetic tests that are generated as a linear combination of two different retry loops.

The amount of critical works in the retry loop 1 and 2 are given at the top and right side of the figures. The latency of parallel work executed after retry loop 1 and 2 are equal. Here, x-axis provides the parallel work latency plus the latency of the small retry loop. This is because we expect no failures in the small retry loop (See Lemma 11), therefore, we assume that its latency is a part of the latency of conflict-free parallel work. The distribution of fails in the retry loops are illustrated and all throughput curves are normalized with a factor of 175 (to be easily seen in the same graph). Fails per success values are not normalized and a success is obtained after completing all retry loops.

Results are compared with the model for the single retry loop case where the single retry loop is equal to the longest retry loop, while the other retry loops are

part of the parallel section. In Figure 2.11, we observe that fails indeed mostly happen in the longest retry loop and our estimates for single retry loop capture the behavior of the linear combination of retry loops.

2.6.3 Treiber's Stack

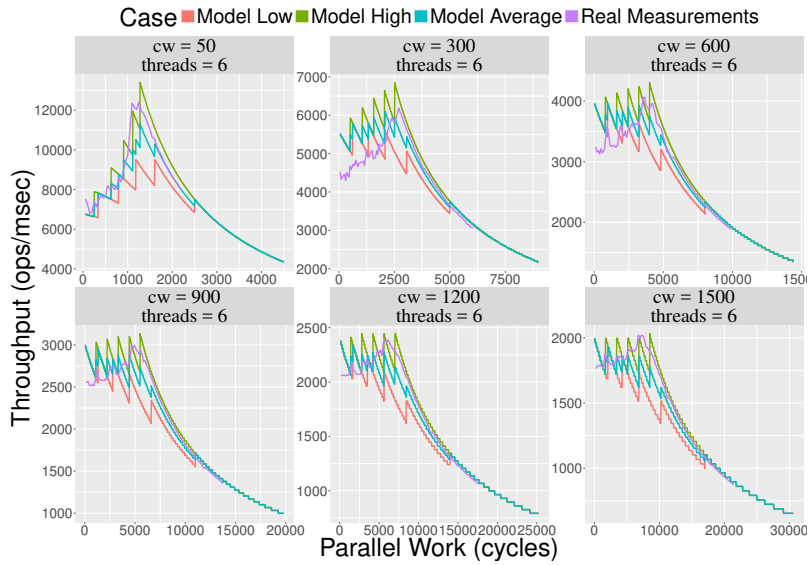


Figure 2.12: Pop on Treiber's stack

The lock-free stack by Treiber [9] is one of the most studied efficient data structures. Pop and Push both contain a retry loop, such that each retry starts with a *Read* and ends with *CAS* on the shared top pointer. In order to validate our model, we start by using Pops. From a stack which is initiated with many elements, threads continuously pop elements for a given amount of time. We count the total number of pop operations per millisecond. Each Pop first reads the top pointer and gets the next pointer of the first element to obtain the address of the second element in the stack, before attempting to *CAS* with the address of the second element. The access to the next pointer of the first element occurs in

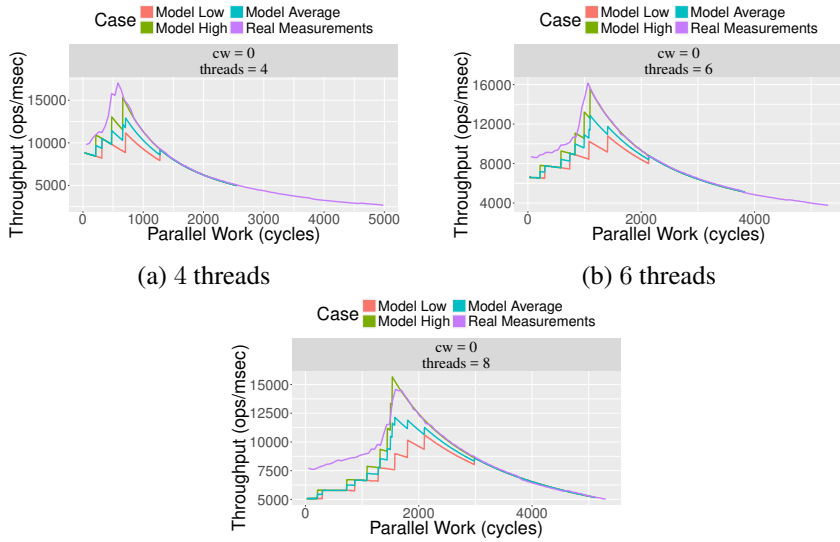
between the *Read* and the *CAS*. Thus, it represents the work in retry loop (cw). This memory access can possibly introduce a costly cache miss depending on the locality of the popped element.

To validate our model with different cw values, we make use of this costly cache miss possibility. We allocate a contiguous chunk of memory and align each element to a cache line. Then, we initialize the stack by pushing elements from contiguous memory either with a single or large stride to disable the prefetcher. When we measure the latency of cw in Pop for single and large stride cases, we obtain the values that are approximately 50 and 300 cycles, respectively. As a remark, 300 cycles is the cost of an L3 miss in our system when it is serviced from the local main memory module. To create more test cases with larger cw, we extended the stack implementation to pop multiple elements with a single operation. Thus, each access to the next element could introduce an additional L3 cache miss while popping multiple elements. By doing so, we created cases in which each thread pops 2, 3, *etc.* elements, and cw goes to 600, 900, *etc.* cycles, respectively.

In Figure 2.12, comparison of the experimental results from Treiber's stack and our model is provided. The trend in the curves are similar to synthetic tests with corresponding cw, except the peak points are slightly higher. This is presumably happening as a result of cache effects which reduce the gaps between successes. More precisely but still speculatively, a thread experiences an L3 cache miss and starts to fetch the data from memory. After a while, another thread experience the same miss on the same memory location but the data is already on its way. This aligns the threads and reduces the gaps between successes. The performance tends to the high estimate curve which represents the case with small gaps that leads to better throughput.

2.6.4 Shared Counter

In [4], the authors have implemented a "scalable statistics counters" relying on the following idea: when contention is low, the implementation is a regular concurrent counter with a *CAS*; when the counter starts to be contended,



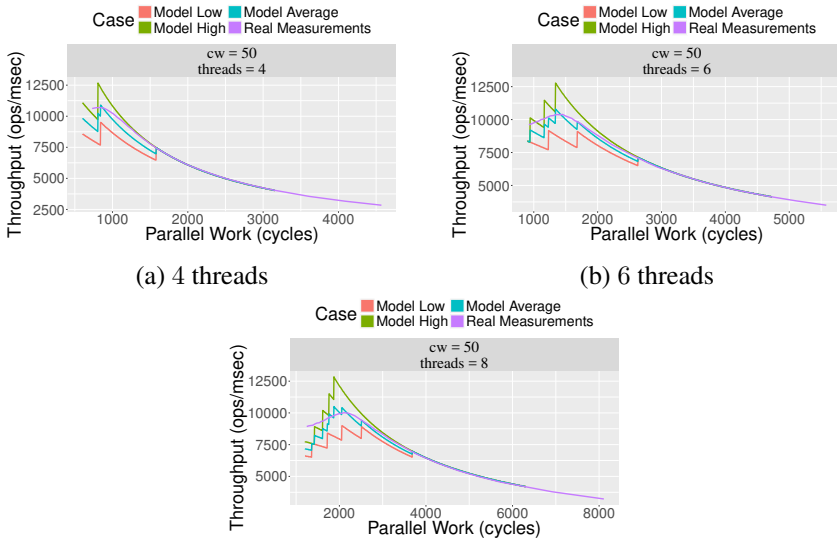
(a) 4 threads

(b) 6 threads

(c) 8 threads

Figure 2.13: Increment on a shared counter

it switches to a statistical implementation, where the counter is actually incremented less frequently, but by a higher value. One key point of this algorithm is the switch point, which is decided thanks to the number of failed increments; our model can be used by providing the peak point of performance of the regular counter implementation as the switch point. We then have implemented a shared counter which is basically a *Fetch-and-Increment* using a *CAS*, and compared it with our analysis. The result is illustrated in Figure 2.13, and shows that the parallel section size corresponding to the peak point is correctly estimated using our analysis. The trend in the curves and their causes are similar to the ones that are mentioned for the synthetic tests.



(a) 4 threads

(b) 6 threads

(c) 8 threads

Figure 2.14: DeleteMin on a priority list

2.6.5 DeleteMin in Priority List

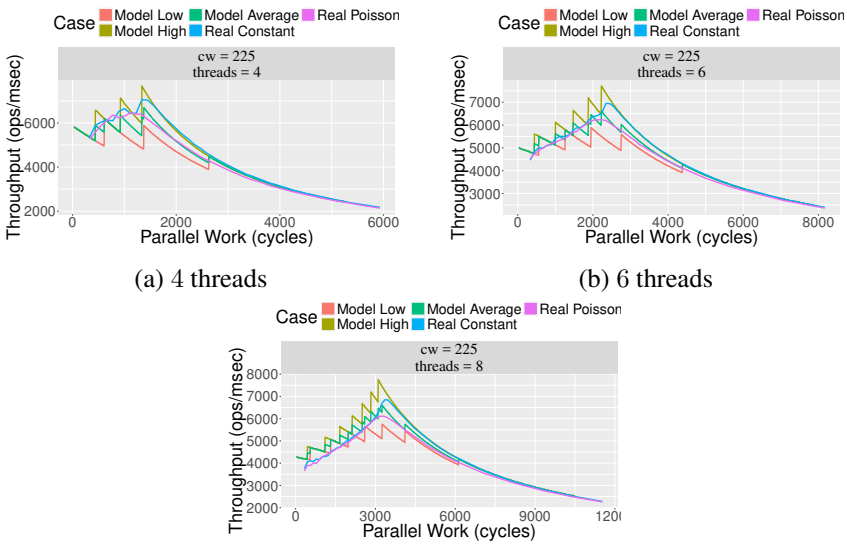
We have applied our model to DeleteMin of the skiplist based priority queue designed in [8]. DeleteMin traverses the list from the beginning of the lowest level, finds the first node that is not logically deleted, and tries to delete it by marking. If the operation does not succeed, it continues with the next node. Physical removal is done in batches when reaching a threshold on the number of deleted prefixes, and is followed by a restructuring of the list by updating the higher level pointers, which is conducted by the thread that is successful in redirecting the head to the node deleted by itself.

We consider the last link traversal before the logical deletion as critical work, as it continues with the next node in case of failure. The rest of the traversal is attributed to the parallel section as the threads can proceed concurrently without interference. We measured the average cost of a traversal under

low contention. In addition, the average cost of restructuring is also included in the parallel section since it is executed infrequently.

We initialize the priority queue with a large set of elements. As illustrated in Figure 2.14, the smallest pw value is not zero as the average cost of traversal and restructuring is intrinsically included. The peak point is in the estimated place but the curve does not go down sharply under high contention. This presumably occurs as the traversal might require more than one steps (link access) after a failed attempt, which creates a back-off effect.

2.6.6 Enqueue-Dequeue on a Queue



(a) 4 threads

(b) 6 threads

(c) 8 threads

Figure 2.15: Enqueue-Dequeue on Michael and Scott queues

In order to demonstrate the validity of the model with several retry loops, and that the results covers a wider spectrum of application and designs from the ones we focused in our model, we studied the following setting: the threads

share a queue, and each thread enqueues an element, executes the parallel section, dequeues an element, and reiterates. We consider the queue implementation by Michael and Scott [1], that is usually viewed as the reference queue while looking at lock-free queue implementations.

Dequeue operations fit immediately into our model but Enqueue operations need an adjustment due to the helping mechanism. Note that without this helping mechanism, a simple queue implementation would fit directly, but we also want to show that the model is malleable, *i.e.* the fundamental behavior remains unchanged even if we divert slightly from the initial assumptions. We consider an equivalent execution that catches up with the model, and use it to approximate the performance of the actual execution of Enqueue.

Enqueue is composed of two steps. Firstly, the new node is attached to the last node of the queue via a *CAS*, that we denote by $\mathsf{CAS_A}$, leading to a transient state. Secondly, the tail is redirected to point to the new node via another *CAS*, that we denote by $\mathsf{CAS_B}$, which brings back the queue into a steady state.

A new Enqueue can not proceed before the two steps of previous success are completed. The first step is the linearization point of operation and the second step could be conducted by a different thread through the helping mechanism. In order to start a new Enqueue, concurrent Enqueues help the completion of the second step of the last success if they find the queue in the transient state. Alternatively, they try to attach their node to the queue if the queue is in the steady state at the instant of check. This process continues until they manage to attach their node to the queue via a retry loop in which state is checked and corresponding CAS is executed.

The flow of an Enqueue is determined by this state checks. Thus, an Enqueue could execute multiple $\mathsf{CAS_B}$ (successful or failing) and multiple $\mathsf{CAS_A}$ (failing) in an interleaved manner, before succeeding in $\mathsf{CAS_A}$ at the end of the last retry. If we assume that both states are equally probable for a check instant which will then end up with a retry, the number of CAS s that ends up with a retry are expected to be distributed equally among $\mathsf{CAS_A}$ and $\mathsf{CAS_B}$ for each thread. In addition, each thread has a successful $\mathsf{CAS_A}$ (which linearizes

the Enqueue) and a $\mathsf{CAS_B}$ at the end of the operation which could either be successful or failed by a concurrent helper thread.

The model can be applied by attributing $\mathsf{CAS_A}$-$\mathsf{CAS_B}$ couple to a single retry loop iteration and represent it as a larger retry loop since the successful couple can not overlap with another successful one and all overlapping ones fail. With a straightforward extension of the expansion formula, we accomodate the $\mathsf{CAS_A}$ in the critical work which can also expand, and use $\mathsf{CAS_B}$ as the *CAS* of our model.

In addition, we take one step further outside the analysis by including a new case, where the parallel section follows a Poisson distribution, instead of being constant. pw is chosen as the mean to generate Poisson distribution instead of taking it constant. The results are illustrated in Figure 2.15. Our model provides good estimates for the constant pw and also reasonable results for the Poisson distribution case, although this case deviates from (/extends) our model assumptions. The advantage of regularity, which brings synchronization to threads, can be observed when the constant and Poisson distributions are compared. In the Poisson distribution, the threads start to fail with larger pw, which smoothes the curve around the peak of the throughput curve.

2.6.7 Discussion

In this subsection we discuss the adequacy of our model, specifically the cyclic argument, to capture the behavior that we observe in practice. Figure 2.16 illustrates the frequency of occurrence of a given number of consecutive fails, together with average fails per success values and the throughput values, normalized by a constant factor so that they can be seen on the graph. In the background, the frequency of occurrence of a given number of consecutive fails before success is presented. As a remark, the frequency of 6+ fails is plotted together with 6. We expect to see a frequency distribution concentrated around the average fails per success value, within the bounds computed by our model.

While comparing the distribution of failures with the throughput, we could conjecture that the bumps come from the fact that the failures spread out. How-

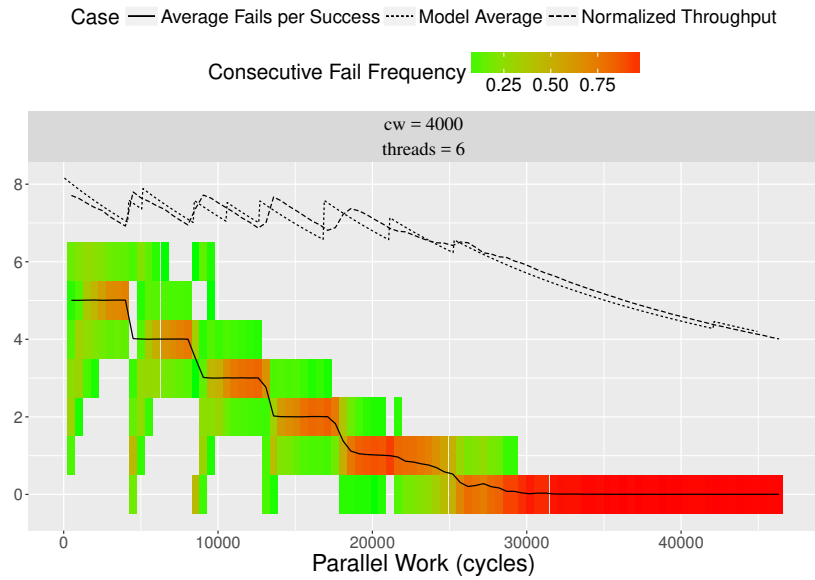
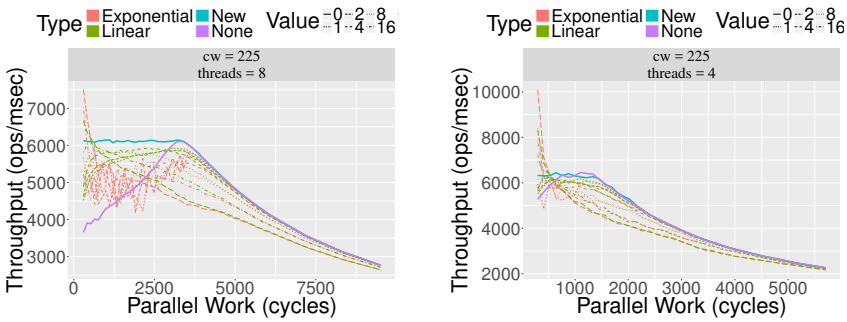


Figure 2.16: Consecutive Fails Frequency

ever, our model captures correctly the throughput variations and thus strips down the right impacting factor. The spread of the distribution of failures indicates the violation of a stable cyclic execution (that takes place in our model), but in these regions, r actually gets close to 0, as well as the minimum of all gaps. The scattering in failures shows that, during the execution, a thread is overtaken by another one. Still, as gaps are close to 0, the imaginary execution, in which we switch the two thread IDs, would create almost the same performance effect. This reasoning is strengthened by the fact that the actual average number of failures follows the step behavior, predicted by our model. This shows that even when the real execution is not cyclic and the distribution of failures is not concentrated, our model that results in a cyclic execution remains a close approximation of the actual execution.

2.6.8 Back-Off Tuning



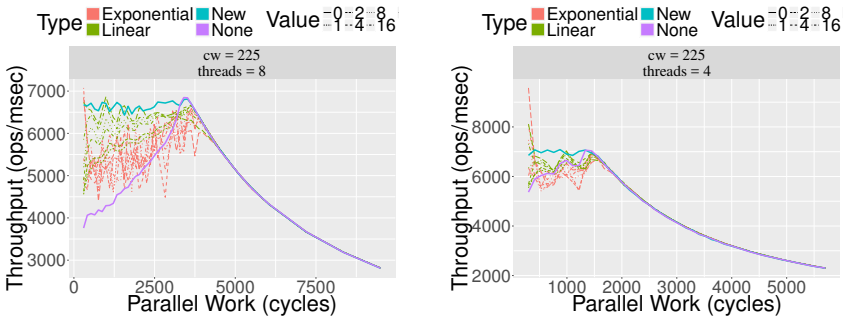
(a) 8 threads (b) 4 threads

Figure 2.17: Comparison of back-off schemes for Poisson Distribution

Together with our analysis comes a natural back-off strategy: we estimate the pw corresponding to the peak point of the average curve, and when the parallel section is smaller than the corresponding pw, we add a back-off in the parallel section, so that the new parallel section is at the peak point.

We have applied exponential, linear and our back-off strategy to the Enqueue/Dequeue experiment specified above. Our back-off estimate provides good results for both types of distribution. In Figure 2.17 (where the values of back-off are steps of 115 cycles), the comparison is plotted for the Poisson distribution, which is likely to be the worst for our back-off. Our back-off strategy is better than the other, except for very small parallel sections, but the other back-off strategies should be tuned for each value of pw.

We obtained the same shapes while removing the distribution law and considering constant values. The results are illustrated in Figure 2.18.



(a) 8 threads

(b) 4 threads

Figure 2.18: Comparison of back-off schemes for constant pw

2.7 Conclusion

In this paper, we have modeled and analyzed the performance of a general class of lock-free algorithms, and have so been able to predict the throughput of such algorithms, on actual system executions. The analysis rely on the estimation of two impacting factors that lower the throughput: on the one hand, the expansion, due to the serialization of the atomic primitives that take place in the retry loops; on the other hand, the wasted retries, due to a non-optimal synchronization between the running threads. We have derived methods to calculate those parameters, along with the final throughput estimate, that is calculated from a combination of these two previous parameters. As a side result of our work, this accurate prediction enables the design of a back-off technique that performs better than other well-known techniques, namely linear and exponential back-offs.

As a future work, we envision to enlarge the domain of validity of the model, in order to cope with data structures whose operations do not have constant retry loop, as well as the framework, so that it includes more various access patterns. The fact that our results extend outside the model we consider allows us to be optimistic on impacting factors introduced in this work. Finally, we

also foresee studying back-off techniques that would combine a back-off in the parallel section (for lower contention) and in the retry loops (for higher robustness).

Bibliography

[1] Maged M. Michael and Michael L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the ACM Symposium on Principles of Distributed Computing (PoDC)*. 1996, pp. 267–275, ACM.

[2] Nir Shavit and Itay Lotan, "Skiplist-based concurrent priority queues," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. 2000, pp. 263–268, IEEE Computer Society.

[3] Danny Hendler, Nir Shavit, and Lena Yerushalmi, "A scalable lock-free stack algorithm," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 70, no. 1, pp. 1–12, 2010.

[4] Dave Dice, Yossi Lev, and Mark Moir, "Scalable statistics counters," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2013, pp. 43–52, ACM.

[5] J. D. Valois, "Implementing Lock-Free Queues," in *Proceedings of International Conference on Parallel and Distributed Systems (ICPADS)*, December 1994, pp. 64–69.

[6] Kristijan Dragicevic and Daniel Bauer, "A survey of concurrent priority queue algorithms," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. 2008, pp. 1–6, IEEE.

[7] Alex Kogan and Maurice Herlihy, "The future(s) of shared data structures," in *Proceedings of the ACM Symposium on Principles of Distributed Computing (PoDC)*. 2014, pp. 30–39, ACM.

[8] Jonatan Lindén and Bengt Jonsson, "A skiplist-based concurrent priority queue with minimal memory contention," in *Proceedings of the International Conference on Principle of Distributed Systems (OPODIS)*. 2013, pp. 206–220, Springer.

[9] R. Kent Treiber, *Systems programming: Coping with parallelism*, International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[10] James H. Anderson, Srikanth Ramamurthy, and Kevin Jeffay, "Real-time computing with lock-free shared objects," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 2, pp. 134–165, 1997.

[11] Intel, "Lock scaling analysis on Intel® Xeon® processors," Tech. Rep. 328878-001, Intel, 2013.

[12] Juan Alemany and Edward W. Felten, "Performance issues in non-blocking synchronization on shared-memory multiprocessors," in *Proceedings of the ACM Symposium on Principles of Distributed Computing (PoDC)*. 1992, pp. 125–134, ACM.

[13] Dan Alistarh, Keren Censor-Hillel, and Nir Shavit, "Are lock-free concurrent algorithms practically wait-free?," in *Proceedings of the ACM Symposium on Theory of Computing (STOC)*. 2014, pp. 714–723, ACM.

[14] James R. Goodman and Herbert Hing Jing Hum, "Mesif: A two-hop cache coherency protocol for point-to-point interconnects," Tech. Rep., University of Auckland, 2009.

[15] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 2013, pp. 33–48, ACM.

RESULT II

Aras Atalar, Paul Renaud-Goud and Philippas Tsigas
How Lock-free Data Structures Perform in Dynamic Environments: Models and Analyses
In the Proceedings of the 20th International Conference on Principles of Distributed Systems, OPODIS 2016
pages 1-17, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik 2016.

3

RESULT II - How Lock-free Data Structures Perform in Dynamic Environments: Models and Analyses

Abstract

In this paper we present two analytical frameworks for calculating the performance of lock-free data structures. Lock-free data structures are based on retry loops and are called by application-specific routines. In contrast to previous work, we consider in this paper lock-free data structures in dynamic environments. The size of each of the retry loops, and the size of the application routines invoked in between, are not constant but may change dynamically. The new frameworks follow two different approaches. The first framework, the sim-

plest one, is based on queuing theory. It introduces an average-based approach that facilitates a more coarse-grained analysis, with the benefit of being ignorant of size distributions. Because of this independence from the distribution nature it covers a set of complicated designs. The second approach, instantiated with an exponential distribution for the size of the application routines, uses Markov chains, and is tighter because it constructs stochastically the execution, step by step.

Both frameworks provide a performance estimate which is close to what we observe in practice. We have validated our analysis on (i) several fundamental lock-free data structures such as stacks, queues, deques and counters, some of them employing helping mechanisms, and (ii) synthetic tests covering a wide range of possible lock-free designs. We show the applicability of our results by introducing new back-off mechanisms, tested in application contexts, and by designing an efficient memory management scheme that typical lock-free algorithms can utilize.

3.1 Introduction

During the last two decades, lock-free data structures have received a lot of attention in the literature, and have been accepted in industrial applications, *e.g.* in the Intel's Threading Building Blocks Framework [1], the Java concurrency package [2] and the Microsoft .NET Framework [3]. Lock-free implementations provide indeed a way out of several limitations of their lock-based counterparts, in robustness, availability and programming flexibility. Last but not least, the advent of multi-core processors has pushed lock-freedom on top of the toolbox for achieving scalable synchronization.

Naturally, the development of lock-free data structures was accompanied by studies on the performance of such data structures, in order to characterize their scalability. Having no guarantee on the execution time of an individual operation, the time complexity analyses of lock-free algorithms have turned towards amortized analyses. The so-called amortized analyses are thus interested in the worst-case behavior over a sequence of operations, which can be seen as

a worst-case bound on the average time per operation. In order to cover various contention environments, the time complexity of the algorithms is often parametrized by different contention measures, such as point [4], interval [5] or step [6] contention. Nonetheless these investigations are targeting worst-case asymptotic behaviors. There is a lack of analytical results in the literature capable of describing the execution of lock-free algorithms on top of a hardware platform, and providing predictions that are close to what is observed in practice. Asymptotic bounds are particularly useful to rank different algorithms, since they rely on a strong theoretical background, but the presence of potentially high constants might produce misleading results. Yet, an absolute prediction of the performance can be of great importance by constituting the first step for further optimizations.

The common measure of performance for data structures is throughput, defined as the number of operations on the data structure per unit of time. To this end, this performance measure is usually obtained by considering an algorithm that strings together a pure sequence of calls to an operation on the data structure. However, when used in a more realistic context, the calls to the operations are mixed with application-specific code (that we call here parallel work). For instance, in a work-stealing environment designed with deques, a thread basically runs one of the following actions: pushing a new-generated task in its deque, popping a task from a deque or executing a task. The modifications on the deques are thus interleaved with deque-independent work. There exist some papers that consider in their experiments local computations between calls to operations during their respective evaluations, but the amount of local computations follows a given distribution varying from paper to paper, *e.g.* constant [7], uniform [8], exponential [9].

In this work, we derive a general approach for unknown distributions of the size of the application-specific code, as well as a tighter method when it follows an exponential distribution.

As for modeling the data structure itself, we use as a basis the universal construction described by Herlihy in [10], where it is shown that any abstract data type can get such a lock-free implementation, which relies on one retry

loop. Moreover, we have particularly focused our experiments on data structures that have inherent sequential bottlenecks (stack, queue, shared counter, deque). Coming back to amortized analyses, the time complexity of an operation is often expressed as a contention-free time complexity added with a contention overhead. In this paper, we want to model and analyze the impact of contention. Loosely speaking, the data structures with inherent sequential bottlenecks have lightweight operations (*i.e.* low contention-free complexity) and they are prone to high contention overheads. In contrast, the data structures that present natural parallelism, or that employ contention alleviation techniques, provide heavyweight operations (*i.e.* high contention-free complexity) and behave differently, compared to the previous ones, under contention. Our analyses examine this trade-off and then facilitate conscious decisions in the data structures design and use.

We propose two different approaches that analyze the performance of such data structures. On the one hand, we derive an average-based approach invoking queuing theory, which provides the throughput of a lock-free algorithm without any knowledge about the distribution of the parallel work. This approach is flexible but allows only a coarse-grained analysis, and hence a partial knowledge of the contention that stresses the data structure. On the other hand, we exhibit a detailed picture of the execution of the algorithm when the parallel work is instantiated with an exponential distribution, through a second complementary approach. We prove that the multi-threaded execution follows a Markovian process and a Markov chain analysis allows us to pursue and reconstruct the execution, and to compute a more accurate throughput.

We finally show several ways to use our analyses and we evaluate the validity of our ideas by experimental results. Those two analysis approaches give a good understanding of the phenomena that drive the performance of a lock-free data structure, at a high-level for the average-based approach, and at a detailed level for the constructive method. Moreover, our results provide a common framework to compare different implementations of a data structure, in a fair manner. We also emphasize that there exist several concrete paths to apply our analyses. To this end, based on the knowledge about the application at

hand, we implement two back-off strategies. We show the applicability of these strategies by tuning a Delaunay triangulation application [11] and a streaming pipeline component which is fed with trade exchange workloads [12]. These experiments reveal the validity of our analyses in the application domain, under non-synthetic workloads and diverse access patterns. We confirm the benefits of our theoretical results by designing a new adaptive memory management mechanism for lock-free data structures in dynamic environments which surpasses the traditional scheme and which is such that the loss in performance, when compared to a static data structure without memory management, is largely leveraged. This memory management mechanism is based on the analyses presented in this paper.

The rest of the paper is organized as follows: we start by presenting related work in Section 5.2, then we define the algorithm and the platform that we consider, together with concepts that are common to our both approaches in Section 3.3. The average-based approach is described in Section 3.4, while the constructive analysis is exposed in Section 3.5, and both methods are evaluated in the experiment part that is presented in Section 3.6.

3.2 Related Work

Approaches that are based on Markov chains and queueing theory, are commonly employed to analyze the performance of parallel programs in concurrent environments. Yu *et al.* [13] have provided an analytical model to estimate the mean transaction completion time for the transactional memory systems. They make use of a continuous-time Markov chain queuing model to analyze the execution of transactions, in which they formulate the state transition rates by considering the arrival rate, the service time for the transactions together with other parameters such as conflict rate that statistically quantifies the spatial (intersecting data set) and temporal (overlapped time) aspects of conflicts. In [14], Al-Bahra has mentioned Little's Law as an appropriate tool to understand the effects of contention on serialized resources for synchronization paradigms.

Closer to our work, Alistarh *et al.* [15] have studied the same class of lock-

free data structures that we consider in this paper. They show initially that the lock-free algorithms are statistically wait-free and going further they exhibit upper bounds on the performance. Their analysis is done in terms of scheduler steps, in a system where only one thread can be scheduled (and can then run) at each step. If compared with execution time, this is particularly appropriate to a system where the instructions of the threads cannot be done in parallel (*e.g.* multi-threaded program on a multi-core processor with only writes on the same cache line of the shared memory). In our paper, the execution is evaluated in terms of processor cycles, strongly related to the execution time. In addition, the "parallel work" and the "critical work" can be done in parallel. Also, they bound the asymptotic expected system latency (with a big O, when the number of threads tends to infinity), while in our paper we estimate the throughput (close to the inverse of system latency) for any number of threads.

Comparing to our previous work: In [16], we illustrate the performance impacting factors and the model we use to cover a subset of lock-free structures that we consider in this paper. In the former paper, the analysis is built upon properties that arise only when the sizes of the critical work and the parallel work are *constant*. There, we show that the execution is not memoryless due to the natural synchrony provided by the retry loops; at the end of the line, we prove that the execution is cyclic and use this property to bound the rate of failed retries.

Here, we provide two new approaches which serve different purposes. In the first approach, we relax the assumptions regarding the critical work and parallel work parameters, that we respectively use to model the data structure operations and the application specific code from which the data structure operations are called. The first approach relies on the expected values of the size of the critical work and the parallel work. This allows us to cover, compared to our previous analysis, more advanced lock-free data structure operations, see Section 3.6.3. Also, we can analyze the data structures running on a larger variety of application specific environments, thanks to the relaxed assumption on the size of the parallel work. The second approach provides a tight analysis when the parallel work follows an exponential distribution. We can observe a

significant decrease in the performance when the parallel work is initiated with exponential distribution in comparison to the cases where the parallel work is constant as in our previous work, see Section 3.6.2.1. The tight analyses, in our previous work and the second approach presented in this paper, reveal for the first time an analytical understanding of this phenomenon.

This paper is complementary to the previous work, not only because of the difference in the analysis tools, the extensive set of data structures and the application specific environments that it considers but also because they together exhibit the impact of the size distributions of the parallel work on the performance of lock-free data structures.

3.3 Preliminaries

We describe in this section the structure of the algorithm that is covered by our model. We explain how to analyze the execution of an instance of such an algorithm when executed by several threads, by slicing this execution into a sequence of adjacent success periods, where a success period is an interval of time during which exactly one operation returns. Each of the success periods is further split into two by the first access to the data structure in the considered retry loop. This execution pattern reflects fundamental phases of both analyses, whose first steps and general direction are outlined at the end of the section.

3.3.1 System Settings

All threads call Procedure AbstractAlgorithm (see Figure 3.1) when they are spawned. So each thread follows a simple though expressive pattern: a sequence of calls to an operation on the data structure, interleaved with some parallel work during which the thread does not try to modify the data structure. For instance, it can represent a work-stealing algorithm, as described in the introduction.

The algorithm is decomposed in two main sections: the *parallel section*, represented on line 2, and the *retry loop* (which represents one operation on the

shared data structure) from line 3 to line 6. A *retry* starts at line 4 and ends at line 6. The outer loop that goes from line 1 to line 6 is designated as the *work loop*.

In each retry, a thread tries to modify the data structure and does not exit the retry loop until it has successfully modified the data structure. It firstly reads the access point AP of the data structure, then, according to the value that has been read, and possibly to other previous computations that occurred in the past, the thread prepares, during the critical work, the new desired value as an access point of the data structure. Finally, it atomically tries to perform the change through a call to the *CAS* primitive. If it succeeds, *i.e.* if the access point has not been changed by another thread between the first *Read* and the *CAS*, then it goes to the next parallel section, otherwise it repeats the process. The retry loop is composed of at least one retry (and the first iteration of the retry loop is strictly speaking not a retry, but a try).

We denote by cc the execution time of a *CAS* when the executing thread does not own the cache line in exclusive mode, in a setting where all threads share a last level cache. Typically, there exists a thread that touches the data between two requests of the same thread, therefore this cost is paid at every occurrence of a *CAS*. As for the *Read*s, rc holds for the execution time of a cache miss. When a thread executes a failed *CAS*, it immediately reads the same cache line (at the beginning of the next retry), so the cache line is not missing, and the execution time of the *Read* is considered as null. However, when the thread comes back from the parallel section, a cache miss is paid. To conclude with the parameters related to the platform, we dispose of P cores, where the *CAS* (resp. the *Read*) latency is identical for all cores, *i.e.* cc (resp. rc) is constant.

The algorithm is parametrized by two execution times. In the general case, the execution time of an occurrence of the parallel section (application-specific section) is a random variable that follows an unknown probability distribution. In the same way, the execution time of the critical work (specific to a data structure) can vary while following an unknown probability distribution. The only provided information is the mean value of those two execution times: cw for the critical work, and pw for the parallel work. These values will be given

in units of work, where $1\,\text{u.o.w.} = 50\,\text{cycles}$.

3.3.2 Execution Description

It has been underlined in [16] that there are two main conflicts that degrade the performance of the data structures which do not offer a great degree of parallelism: logical and hardware conflicts.

Logical conflicts occur when there are more than one thread in the retry loop at a given time (happens typically when the number of threads is high or when the parallel section is small). At any time, considering only the threads that are in the retry loop, there is indeed at most one thread whose retry will be successful (*i.e.* whose ending *CAS* will succeed), which implies the execution of more retries for the failing threads. In addition, after a thread executes successfully its final *CAS*, the other threads of the retry loop have first to finish their current retry before starting a potentially successful retry, since they are not informed yet that their current retry is doomed to failure. This creates some "holes" in the execution where all threads are executing useless work.

The threads will also experience *hardware conflicts*: if several threads are requesting for the same data, so that they can operate a *CAS* on it, a single thread will be satisfied. All the other threads will have to wait until the current *CAS* is finished, and give a new try when this *CAS* is done. While waiting for the ownership of the cache line, the requesting threads cannot perform any useful work. This waiting time is referred to as *expansion*.

We now refine the description of the execution of the algorithm. The timeline is initially decomposed into a sequence of success periods that will define the throughput. A success period is an interval of time of the execution that (i) starts after a successful *CAS*, (ii) contains a single successful *CAS*, (iii) finishes after this successful *CAS*. As explained in the previous subsection, to be successful in its retry, a thread has first to access the data structure, then modify it locally, and finally execute a *CAS*, while no other thread performs changes on the data structure. That is why each success period is further cut into two main phases (see Figure 3.2). During the first phase, whose duration is called the

```
Procedure AbstractAlgorithm
1 while ! done do
2     Parallel_Work();
3     while ! success do
4         current ← Read(AP);
5         new ← Critical_Work(current);
6         success ← CAS(AP, current, new);
```

Figure 3.1: Thread procedure

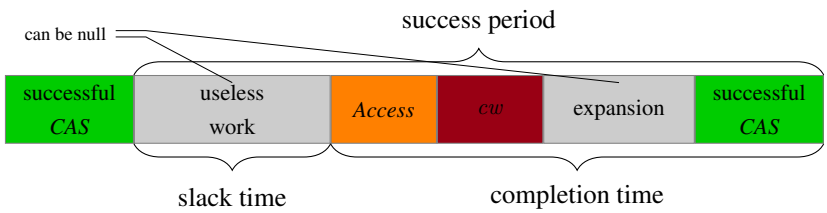


Figure 3.2: Success period

slack time, no thread is accessing the data structure. The second phase, characterized by the *completion time*, starts with the first access to the data structure (by any thread). Note that this *Access* could be either a *Read* (if the concerned thread just exited the parallel section) or a failed *CAS* (if the thread was already in the retry loop). The next successful *CAS* will come at least after cw (one thread has to traverse the critical work anyway), that is why we split the latter phase into: cw, then expansion, and finally a successful *CAS*.

3.3.3 Our Approaches

In this work, we propose two different approaches to compute the throughput of a lock-free algorithm, which we name as average-based and constructive. The average-based approach relies on queuing theory and is focused on the average

behavior of the algorithm: the throughput is obtained through the computation of the expectation of the success period at a random time. As for the constructive approach, it describes precisely the instants of accesses and modifications to the data structure in each success period: in this way, we are able to deconstruct and reconstruct the execution, according to observed events. The constructive approach leads to a more accurate prediction at the expense of requiring more information about the algorithm: the distribution functions of the critical and parallel works have indeed to be instantiated.

In both cases, we partition the domain space into different levels of contention (or *modes*); these partitions are independent across approaches, even if we expect similarities, but in each case, cover the whole domain space (all values of critical work, parallel work and number of threads).

3.3.3.1 Average-based Analysis

We distinguish two main modes in which the algorithm can run: contended and non-contended. In the non-contended mode, *i.e.* when the parallel work is large or the number of threads is low, concurrent operations are not likely to collide. So every retry loop will count a single retry, and atomic primitives will not delay each other. In the contended mode, any operation is likely to experience unsuccessful retries before succeeding (logical conflicts), and a retry will last longer than in the non-contended mode because of the collision of atomic primitives (hardware conflicts).

Once all the parameters are given, the analysis is centered around the calculation of a single variable $\overline{P_{rl}}$, which represents the expectation of the number of threads inside the retry loop at a random instant. Based on this variable, we are able to express the expected expansion $\overline{e}\left(\overline{P_{rl}}\right)$ at a random time. As a next step, we show how this expansion can be used to estimate the expected slack time $\overline{st}\left(\overline{P_{rl}}\right)$ and the expected completion time $\overline{ct}\left(\overline{P_{rl}}\right)$, and at the end, the expected time of a success period $\overline{sp}\left(\overline{P_{rl}}\right)$.

3.3.3.2 Constructive Method

The previous average-based reasoning is founded on expected values at a random time, while in the constructive approach, we study each success period individually, based on the number of threads at the beginning of the considered success period. So we are able to exhibit more clearly the instants of occurrences of the different accesses and modifications to the data structure, and thus to predict the throughput more accurately.

We rely on the same set of values used in the average-based approach, but these values are now associated with a given success period. Thus the number of threads inside the retry loop P_{rl}, as well as the slack time and the completion time are evaluated at the beginning of each success period. We denote these times in the same way as in the first approach, but remove the bar on top since these values are not expectations any more.

The different contention modes do not characterize here the steady-state of the data structure as in the previous approach but are associated with the current success period. Accordingly, the contention can oscillate through different modes in the course of the execution. First, a success period is not contended when $P_{rl} = 0$, *i.e.* when there is no thread in the retry loop after a successful *CAS*. In this case, the first thread that exits the parallel section will be successful, and the *Access* of the sequence will be a *Read*. Second, the contention of a success period is high when at any time during the success period, there exists a thread that is executing a *CAS*. In other words, at the end of each *CAS*, there is at least one thread that is waiting for the cache line to operate a *CAS* on it. This implies that the first access of the success period is a *CAS* and occurs immediately after the preceding successful *CAS*: the slack time is null. Third, the mid-contention mode takes place when $P_{rl} > 0$, while at the same time, there are not enough requesting threads to fill the whole success period with *CAS*'s (which implies a non-null slack time). Since these requesting threads have synchronized in the previous success period, *CAS*'s do not collide in the current success period, and because of that, the expansion is null.

3.4 Average-based Approach

We propose in this section our coarse-grained analysis to predict the performance of lock-free data structures. Our approach utilizes fundamental queuing theory techniques, describing the average behavior of the algorithm. In turn, we need only a minimal knowledge about the algorithm: the mean execution time values cw and pw. As explained in Section 3.3.3.1, the system runs in one of the two possible modes: either contended or uncontended.

3.4.1 Contended System

We first consider a system that is contended. When the system is contended, we use Little's law to obtain, at a random time, the expectation of the success period, which is the interval of time between the last and the next successful *CAS*'s (see Figure 3.2).

The stable system that we observe is the parallel section: threads are entering it (after exiting a successful retry loop) at an average rate, stay inside, then leave (while entering a new retry loop). The average number of threads inside the parallel section is $\overline{P_{ps}} = P - \overline{P_{rl}}$, each thread stays for an average duration of pw, and in average, one thread is exiting the retry loop every success period $\overline{sp}\left(\overline{P_{rl}}\right)$, by definition of the success period. According to Little's law [17], we have:

$$\overline{P_{ps}} = pw \times \frac{1}{\overline{sp}\left(\overline{P_{rl}}\right)}, \textit{ i.e.}$$

$$\frac{1}{pw} \times \overline{sp}\left(\overline{P_{rl}}\right) = \frac{1}{P - \overline{P_{rl}}} \tag{3.1}$$

As explained in Section 3.3.2, we further decompose a success period into two parts, separated by the first access to the data structure after a successful *CAS*. We can then write the average success period as the sum of: (i) the expected time before some thread starts its *Access* (the slack time), and (ii) the expected completion time. We compute these two expectations independently and gather them into the success period thanks to:

$$\overline{sp}\left(\overline{P_{rl}}\right) = \overline{st}\left(\overline{P_{rl}}\right) + \overline{ct}\left(\overline{P_{rl}}\right). \tag{3.2}$$

When the data structure is contended, a thread is likely to be successful after some failed retries. Therefore a thread that is successful was already in the retry loop when the previous successful *CAS* occurred. This implies that the *Access* to the data structure will be due to a failed *CAS*, instead of a *Read*.The time before a thread starts its *Access* is then the time before a thread finishes its current critical work since there is a thread currently executing a *CAS*.

3.4.1.1 Expected Completion time

Since the data structure is contended, numerous threads are inside the retry loop, and, due to hardware conflicts, a retry can experience expansion: the more threads inside the retry loop, the longer time between a *CAS* request and the actual execution of this *CAS*. The expectation of the completion time can be written as:

$$\overline{ct}\left(\overline{P_{rl}}\right) = cc + cw + \overline{e}\left(\overline{P_{rl}}\right) + cc, \tag{3.3}$$

where $\overline{e}\left(\overline{P_{rl}}\right)$ is the expectation of expansion when there are $\overline{P_{rl}}$ threads inside the retry loop, in expectation. This expansion can be computed in the same way as in [16], through the following differential equation:

$$\begin{cases} \overline{e}'\left(\overline{P_{rl}}\right) & = & cc \times \dfrac{\frac{cc}{2} + \overline{e}\left(\overline{P_{rl}}\right)}{cc + cw + cc + \overline{e}\left(\overline{P_{rl}}\right)} \\ \overline{e}\left(1\right) & = & 0 \end{cases},$$

by assuming that the expansion starts as soon as strictly more than 1 thread are in the retry loop, in expectation.

3.4.1.2 Expected Slack Time

Concerning the slack time, we consider that, at any time, the threads that are running the retry loop have the same probability to be anywhere in their current retry. However, when a thread is currently executing a *CAS*, the other threads cannot execute as well a *CAS*. The other threads are thus in their critical work

or expansion. For every thread, the time before accessing the data structure is then uniformly distributed between 0 and $cw + \overline{e}\left(\overline{P_{rl}}\right)$.

According to Lemma 12, we conclude that

$$\overline{st}\left(\overline{P_{rl}}\right) = \left(cw + \overline{e}\left(\overline{P_{rl}}\right)\right)/(\overline{P_{rl}} + 1). \tag{3.4}$$

Lemma 12. *Let an integer n, a real positive number a, and n independent random variables $X_1, X_2, \ldots, X_n$, uniformly distributed within $[0, a[$. Let then X be the random variable defined by: $X = \min_{i \in [\![1,n]\!]} X_i$. The expectation of X is:*

$$\mathbb{E}(X) = \frac{a}{n+1}.$$

Proof. Let a positive real number x be such that $x < a$. We have

$$\begin{aligned}
\mathbb{P}(X > x) &= \mathbb{P}(\forall i : X_i > x) \\
&= \prod_{i=1}^{n} \mathbb{P}(X_i > x) \\
\mathbb{P}(X > x) &= \left(\frac{a-x}{a}\right)^n
\end{aligned}$$

Therefore, the probability distribution of X is given by:

$$t \mapsto \frac{n}{a}\left(\frac{a-x}{a}\right)^{n-1},$$

and its expectation is computed through

$$\begin{aligned}
\mathbb{E}(X) &= \frac{n}{a}\int_0^a x \times \left(\frac{a-x}{a}\right)^{n-1} dx \\
&= \frac{n}{a}\int_0^a (a-u) \times \left(\frac{u}{a}\right)^{n-1} du \\
&= \frac{n}{a^n}\int_0^a (a-u) \times u^{n-1}\, du \\
&= \frac{n}{a^n}\left(a \times \frac{a^n}{n} - \frac{a^{n+1}}{n+1}\right) \\
\mathbb{E}(X) &= \frac{a}{n+1}.
\end{aligned}$$

□

3.4.1.3 Expected Success Period

We just have to combine Equations 3.2, 3.3, and 3.4 to obtain the general expression of the expected success period under contention:

$$\overline{sp}\left(\overline{P_{rl}}\right) = \left(1 + \frac{1}{\overline{P_{rl}}+1}\right)\left(cw + \overline{e}\left(\overline{P_{rl}}\right)\right) + 2cc,$$

which leads, according to Equation 3.1, to

$$\frac{1}{pw} \times \left(\frac{\overline{P_{rl}}+2}{\overline{P_{rl}}+1}\left(cw + \overline{e}\left(\overline{P_{rl}}\right)\right) + 2cc\right) = \frac{1}{P - \overline{P_{rl}}}. \tag{3.5}$$

3.4.2 Non-contended System

When the system is not contended, logical conflicts are not likely to happen, hence each thread succeeds in its retry loop at its first *retry*. *A fortiori*, no hardware conflict occurs. Each thread still performs one success every work loop, and the success period is given by

$$\overline{sp}\left(\overline{P_{rl}}\right) = \frac{pw + rc + cw + cc}{P}. \tag{3.6}$$

Moreover, a thread spends in average $rc + cw + cc$ units of time in the retry loop within each work loop. As this holds for every thread, we can obtain the

following expression for the total average number of threads inside the retry loop:

$$\overline{P_{rl}} = \frac{rc + cw + cc}{pw + rc + cw + cc} \times P = \frac{rc + cw + cc}{\overline{sp}\left(\overline{P_{rl}}\right)} \tag{3.7}$$

Equation 3.6 also gives $rc + cw + cc = P \times \overline{sp}\left(\overline{P_{rl}}\right) - pw$, hence, thanks to Equation 3.7,

$$\overline{P_{rl}} = \frac{P \times \overline{sp}\left(\overline{P_{rl}}\right) - pw}{\overline{sp}\left(\overline{P_{rl}}\right)}, \textit{ i.e. } \quad \frac{\overline{sp}\left(\overline{P_{rl}}\right)}{pw} = \frac{1}{P - \overline{P_{rl}}}, \tag{3.8}$$

where $\overline{sp}\left(\overline{P_{rl}}\right) = \frac{rc+cw+cc}{\overline{P_{rl}}}$.

3.4.3 Unified Solving

It remains to decide whenever the data structure is under contention or not, and to find the corresponding solution. Concerning the frontier between contended and non-contended system, we can remark that Equations 3.5 and 3.8 are equivalent if and only if

$$\frac{rc + cw + cc}{\overline{P_{rl}}} = \frac{\overline{P_{rl}} + 2}{\overline{P_{rl}} + 1}\left(cw + \overline{e}\left(\overline{P_{rl}}\right)\right) + 2cc, \tag{3.9}$$

which leads to Lemma 13.

Lemma 13. *The system switches from being non-contended to being contended at* $\overline{P_{rl}} = P_{rl}^{(0)}$*, where*

$$P_{rl}^{(0)} = \frac{-(cc + cw - rc) + \sqrt{(cc + cw - rc)^2 + 4(rc + cw + cc)(cw + 2cc)}}{2(cw + 2cc)}.$$

Proof. We show that:

- $P_{rl}^{(0)}$ is the unique positive solution of Equation 3.9 if the expansion is set to 0,
- $P_{rl}^{(0)} \leq 1$,

- there is no solution of Equation 3.9 with a non-null expansion.

If the expansion is set to 0, then Equation 3.9 can be turned into the second order equation

$$\overline{P_{rl}}^2(cw+2cc)+\overline{P_{rl}}\,(cw+cc-rc)-(rc+cw+cc)=0,$$

that has a single positive solution: $P_{rl}^{(0)}$.

While instantiating the binomial with $\overline{P_{rl}}=1$, we obtain $cw+2(cc-rc)$, which is not negative, since $cc \geq rc$ in all the architectures that we are aware of. As the second order equation has also a negative solution, and $cw+2cc$ is positive, we have that $1 \geq P_{rl}^{(0)}$. This implies that $P_{rl}^{(0)}$ is a solution of the former Equation 3.9: the expansion is indeed a non-decreasing function, thus $0 \leq \overline{e}\left(P_{rl}^{(0)}\right) \leq \overline{e}\,(1)=0$. Still we could have other solutions with a non-null expansion.

However, Equation 3.9 can be rewritten as:

$$rc+cw+cc=\frac{\overline{P_{rl}}+2}{\overline{P_{rl}}+1}\times\overline{P_{rl}}\times\left(cw+\overline{e}\left(\overline{P_{rl}}\right)\right)+2cc. \tag{3.10}$$

The left-hand side of Equation 3.10 is constant, while the right-hand side is increasing, which discards any other solution, hence the lemma. □

Thanks to Lemma 13, we can unify the success period as:

$$\overline{sp}\left(\overline{P_{rl}}\right)=\begin{cases}(rc+cw+cc)/\overline{P_{rl}} & \text{if } \overline{P_{rl}} \leq P_{rl}^{(0)}\\ \left(cw+\overline{e}\left(\overline{P_{rl}}\right)\right)\times\frac{\overline{P_{rl}}+2}{\overline{P_{rl}}+1}+2cc & \text{otherwise.}\end{cases}$$

The unified success period obeys to the following equation

$$\overline{sp}\left(\overline{P_{rl}}\right)=\frac{pw}{P-\overline{P_{rl}}}. \tag{3.11}$$

We show in the following theorem how to compute the throughput estimate; the proof manipulates equations in order to be able to use the fixed-point Knaster-Tarski theorem.

Theorem 5. *The throughput can be obtained iteratively through a fixed-point search, as* $T = \left(\overline{sp}\left(\lim_{n\to+\infty} u_n\right)\right)^{-1}$*, where*

$$\begin{cases} u_0 = \frac{rc+cw+cc}{pw+rc+cw+cc} \times P \\ u_{n+1} = \frac{u_n}{pw+u_n\overline{sp}(u_n)} \times P \qquad \textit{for all } n \geq 0. \end{cases}$$

Proof. Let us note $\mathrm{f}_1\left(\overline{P_{rl}}\right) = \overline{sp}\left(\overline{P_{rl}}\right) \times \overline{P_{rl}}$ and $\mathrm{f}_2\left(\overline{P_{rl}}\right) = pw \times \overline{P_{rl}}/(P - \overline{P_{rl}})$; then Equation 3.11 is equivalent to $\mathrm{f}_1\left(\overline{P_{rl}}\right) = \mathrm{f}_2\left(\overline{P_{rl}}\right)$, and we have some properties on f_1 and f_2.

Firstly, since $x \mapsto x(x+2)/(x+1)$ is non-decreasing on $[0, +\infty[$, as well as the expected expansion, we know that f_1 is a non-decreasing function. Secondly, f_2 is increasing on $[0, P[$, and is bijective from $[0, P[$ to $[0, +\infty[$. We can thus rewrite Equation 3.11 as:

$$\overline{P_{rl}} = {\mathrm{f}_2}^{-1}\left(\mathrm{f}_1\left(\overline{P_{rl}}\right)\right). \tag{3.12}$$

Moreover, ${\mathrm{f}_2}^{-1} \circ \mathrm{f}_1$ is a non-decreasing function, as a composition of two non-decreasing functions. Thirdly, ${\mathrm{f}_2}^{-1}$ can be obtained through $x = \mathrm{f}_2\left({\mathrm{f}_2}^{-1}(x)\right) = pw \times {\mathrm{f}_2}^{-1}(x)/(P - {\mathrm{f}_2}^{-1}(x))$, which leads to

$${\mathrm{f}_2}^{-1}(x) = \frac{x}{pw+x}P.$$

In addition, we know by construction that if $\overline{P_{rl}} > P_{rl}^{(0)}$, then

$$\left(cw + \overline{e}\left(\overline{P_{rl}}\right)\right) \times \frac{\overline{P_{rl}}+2}{\overline{P_{rl}}+1} + 2cc \geq \frac{rc+cw+cc}{\overline{P_{rl}}}. \tag{3.13}$$

Indeed, on the one hand,

$$\lim_{\overline{P_{rl}}\to 0^+} \frac{rc+cw+cc}{\overline{P_{rl}}} = +\infty,$$

and on the other hand, $(cw + \overline{e}\left(\overline{P_{rl}}\right)) \times (\overline{P_{rl}}+2)/(\overline{P_{rl}}+1) + 2cc$ remains bounded. According to Lemma 13, those two functions cross only once, hence Equation 3.13.

Since $\overline{sp}\left(\overline{P_{rl}}\right) = (rc+cw+cc)/\overline{P_{rl}}$ if $\overline{P_{rl}} \leq P_{rl}^{(0)}$, we have $\overline{sp}\left(\overline{P_{rl}}\right) \geq (rc+cw+cc)/\overline{P_{rl}}$ for any $\overline{P_{rl}}$, and then

$$\mathrm{f}_1\left(\overline{P_{rl}}\right) \geq rc+cw+cc.$$

Let then

$$P_{rl}^{(\mathrm{i})} = \frac{rc + cw + cc}{pw + rc + cw + cc} P.$$

We have seen that ${\mathrm{f}_2}^{-1} \circ \mathrm{f}_1$ is a non-decreasing function, hence

$$\begin{aligned} {\mathrm{f}_2}^{-1} \left(\mathrm{f}_1 \left(P_{rl}^{(\mathrm{i})} \right) \right) &\geq {\mathrm{f}_2}^{-1} \left(rc + cw + cc \right) \\ &\geq \frac{rc + cw + cc}{pw + rc + cw + cc} \times P \\ {\mathrm{f}_2}^{-1} \left(\mathrm{f}_1 \left(P_{rl}^{(\mathrm{i})} \right) \right) &\geq P_{rl}^{(\mathrm{i})}. \end{aligned}$$

Since ${\mathrm{f}_2}^{-1}$ is bounded, Equation 3.12 admits a solution.

We are interested in the solution whose $\overline{P_{rl}}$ is minimal since it corresponds to the first attained solution when the expansion grows, starting from 0. The current theorem comes then from the application of the Knaster-Tarski theorem.

□

3.5 Constructive Approach

In this section, we instantiate the probability distribution of the parallel work with an exponential distribution. We have therefore a better knowledge of the behavior of the algorithm, particularly in medium contention cases, which allows us to follow a fine-grained approach that studies individually each successful operation together with every *CAS* occurrence. We provide an elegant and efficient solution that relies on a Markov chain analysis.

3.5.1 Process

We have seen in Section 3.3.3.2 that we split the contention domain into three modes: no contention, medium contention or high contention. The main idea is to start from a configuration with a given number of threads P_{rl} just after a successful *CAS*, and describe what will happen until the next successful *CAS*: what will be the mode of the next success period, and even more precisely, which will be the number of threads at the beginning of the next success period.

As a basis, we consider the execution that would occur without any other thread exiting the parallel section (then entering the retry loop); we call this execution the *internal execution*. This execution follows the success period pattern described in Figure 3.2 (with an infinite slack time if the system is not contended). On top of this basic success period, we inject the threads that can exit the parallel section, which has a double impact. On the one hand, they increase the number of threads inside the retry loop for the next success period. On the other hand, if the first thread that exits the parallel section starts its retry during the slack time of the success period of the internal execution, then this thread will succeed its *Access*, which is a *Read*, and will shrink the actual slack time of the current success period.

According to the distribution probability of the arrival of the new threads, we can compute the probability for the next success period to start with any number of threads. The expression of this stochastic sequence of success periods in terms of Markov chains results in the throughput estimate.

3.5.2 Expansion

The expansion, as before, represents the additional time in the execution time of a retry, due to the serialization of atomic primitives. However, in contrary to Section 3.4.1.1, we compute here this additional time in the current success period, according to the number of threads P_{rl} inside the retry loop at the beginning of the success period. The expansion only appears when the success period is highly contended, *i.e.* when we can find a continuous sequence of *CAS*'s all through the success period.

The expansion is highly correlated with the way the cache coherence protocol handles the exchange of cache lines between threads. We rely on the experiments of the research report associated with [15], which show that if several threads request for the same cache line in order to operate a *CAS*, while another thread is currently executing a *CAS*, they all have an equal probability to obtain the cache line when the current *CAS* is over.

We draw an illustrative example in Figure 3.3. The green *CAS*'s are success-

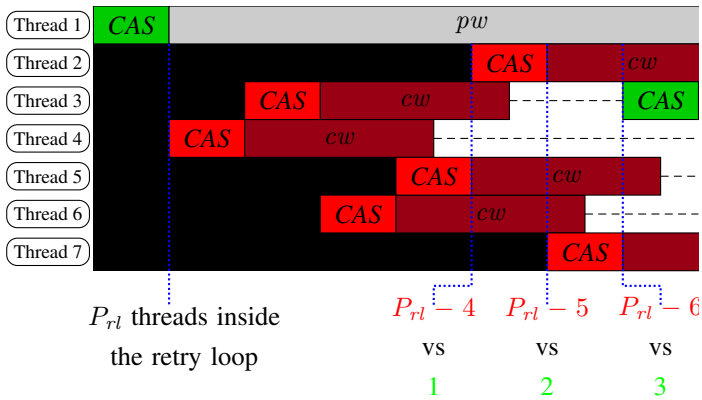


Figure 3.3: Highly-contended execution

ful while the red *CAS*'s fail. To lighten the picture, we hide what happened for the threads before they experience a failed *CAS*. The horizontal dash lines represent the time where a thread wants to access the data in order to operate a *CAS* but has to wait because another thread owns the data in exclusive mode. We can observe in this example that the first thread that accesses the data structure is not the thread whose operation returns.

We are given that P_{rl} threads are inside the retry loop at the end of the previous successful *CAS*, and we only consider those threads. When such a thread executes a *CAS* for the first time, this *CAS* is unsuccessful. The thread was in the retry loop when the successful *CAS* has been executed, so it has read a value that is not up-to-date anymore. However, this failed *CAS* will bring the current version of the value (to compare-and-swap) to the thread, a value that will be up-to-date until a successful *CAS* occurs.

So we have firstly a sequence of failed *CAS*'s until the first thread that operated its *CAS* within the current success period finishes its critical work. At this point, there exists a thread that is executing a *CAS*. When this *CAS* is finished, some threads compete to obtain the cache line. We have two bags of competing threads: in the first bag, the thread that just ended its critical work is alone,

while in the second bag, there are all the threads that were in the retry loop at the beginning of the success period, and did not operate a *CAS* yet. The other, non-competing, threads are running their critical work and do not yet want to access the data.

As described before, every thread has the same probability to become the next owner of the cache line. If a thread from the first bag is drawn, then the *CAS* will be successful and the success period ends. Otherwise, the *CAS* is a failure, and we iterate at the end of this failed *CAS*. However, the thread that just failed its *CAS* is now executing its critical work, and does not request for a new *CAS* until this work has been done, thus it is not anymore in the second bag. In addition, the thread that had executed its *CAS* after the thread of the first bag is now back from its critical work and falls into the first bag. The process iterates until a thread is drawn from the first bag.

As a remark, note that we do not consider threads that are not in the retry loop at the beginning of the success period since even if they come back from the parallel section during the success period, their *Read* will be delayed and their *CAS* is likely to occur after the end of the success period.

Theorem 6 gives the explicit formula for the expansion, based on the previous explanations.

Theorem 6. *The expected time between the end of the critical work of the first thread that operates a CAS in the success period and the beginning of a successful CAS is given by:*

$$e\left(P_{rl}\right) = \lceil cw/cc \rceil cc - cw + \sum_{i=1}^{P_{com}} \frac{i(i-1)}{\left(P_{com}\right)^i} \frac{\left(P_{com}-1\right)!}{\left(P_{com}-i\right)!} \times cc,$$

where $P_{com} = P_{rl} - \lceil cw/cc \rceil$.

Proof. Let us set the timeline so that at the beginning of the success period, *i.e.* just after a successful *CAS*, we are at $t = 0$. Firstly, a success cannot start before $t = t_0$, where $t_0 = cc + \lceil cw/cc \rceil cc$. The quickest thread indeed starts a failed *CAS* at $t = 0$ and comes back from critical work at $t = cc + cw$. It has then to wait for the current *CAS* to finish before being able to obtain the cache line.

At $t = t_0$, $P_{rl} - t_0/cc + 1$ threads are competing for the data. Among them, 1 thread will lead to a successful *CAS*, while the $P_{rl} - t_0/cc$ other threads will end up with a failed *CAS*. If a failed *CAS* occurs, then at $t = t_0 + cc$, the same number of threads compete, but now there is one more potential success and one less potential failure. In the worst case, it will continue until all competing threads will lead to a successful *CAS*.

Let $P_{com} = P_{rl} - t_0/cc + 1$ the number of threads that are competing at each round, and let, for all $i \in [\![1, P_{com}]\!]$, $p_i = i/P_{com}$ the probability to draw a thread that will execute a successful *CAS*.

The expected number of failed *CAS*'s that occurs after the first thread comes back is then given by

$$\begin{aligned}\mathbb{E}(F) = {} & p_1 \times 0 + (1-p_1)p_2 \times 1 + \cdots + \\ & (1-p_1)(1-p_2) \times \cdots \times (1-p_{P_{com}-1}) \times p_{P_{com}} \times (P_{com}-1).\end{aligned}$$

More formally,

$$\begin{aligned}\mathbb{E}(F) &= \sum_{i=1}^{P_{com}} \prod_{j=1}^{i-1} (1-p_j)p_i \times (i-1) \\ &= \sum_{i=1}^{P_{com}} \prod_{j=1}^{i-1} (1-\frac{j}{P_{com}})\frac{i}{P_{com}} \times (i-1) \\ &= \sum_{i=1}^{P_{com}} \frac{1}{(P_{com})^i} \prod_{j=1}^{i-1} (P_{com}-j)i(i-1) \\ \mathbb{E}(F) &= \sum_{i=1}^{P_{com}} \frac{i(i-1)}{(P_{com})^i} \frac{(P_{com}-1)!}{(P_{com}-i)!}\end{aligned}$$

□

3.5.3 Formalization

The parallel work follows an exponential distribution, whose mean is pw. More precisely, if a thread starts a parallel section at the instant t_1, the probability

distribution of the execution time of the parallel section is

$$t \mapsto \lambda e^{-\lambda(t-t_1)} \mathbb{1}_{[t_1,+\infty[}(t)\,, \text{ where } \lambda = \frac{1}{pw}.$$

This probability distribution is memoryless, which implies that the threads that are executing their parallel section cannot be differentiated: at a given instant, the probability distribution of the remaining execution time is the same for all threads in the parallel section, regardless of when the parallel section began. For all threads, it is defined by:

$$t \mapsto \lambda e^{-\lambda t}, \text{ where } \lambda = \frac{1}{pw}.$$

For the behavior in the retry loop, we rely on the same approximation as in the previous section, *i.e.* when a successful thread exits its retry loop, the remaining execution time of the retry of every other thread that is still in the retry loop is uniformly distributed between 0 and the execution time of a whole retry. We have seen that the expectation of this remaining time is the size of the execution time of a retry divided by the number of threads inside the retry loop plus one. Here, we assume that a thread will start a retry at this time. This implies another kind of memoryless property: the behavior of a thread that is in the retry loop does not depend on the moment that it entered its retry loop.

To tackle the problem of estimating the throughput of such a system, we use an approach based on Markov chains. We study the behavior of the system over time, step by step: a state of the Markov chain represents the state of the system when the current success period began (*i.e.* just after a successful *CAS*) and (thus) the system changes state at the end of every successful *CAS*. According to the current state, we are able to compute the probability to reach any other state at the beginning of the next success period. In addition, the two memoryless properties render the description of a state easy to achieve: the number of threads inside the retry loop when the current success begins, indeed fully characterizes the system.

We recall that P_{rl} is the number of threads inside the retry loop when the success period begins. The Markov chain is strongly connected with P_{rl}, since it is composed of P states $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_{P-1}$, where, for all $i \in [\![0, P-1]\!]$,

the success period is in state $\mathcal{S}_i$ iff $P_{rl} = i$. For all $(i,j) \in [\![0, P-1]\!]^2$, $\mathbb{P}(\mathcal{S}_i \to \mathcal{S}_j)$ denotes the probability that a success characterized by $\mathcal{S}_j$ follows a success in state $\mathcal{S}_i$. $st(\mathcal{S}_i \to \mathcal{S}_j)$ denotes the slack time that passed while the system has gone from state $\mathcal{S}_i$ to state $\mathcal{S}_j$. This slack time can be expressed based on the slack time $st(i)$ of the internal execution, *i.e.* the execution that involves only the i threads of the retry loop and ignores the other threads (see Section 3.5.1). Recall that we consider that the slack time of the internal execution with 0 thread is infinite, since no thread will access the data structure. In the same way, we denote by $ct(i)$ the completion time of the internal execution, hence $ct(i) = cc + cw + e(i) + cc$.

We have seen that the level of contention (mode) is determined by P_{rl}, hence the interval $[\![0, P-1]\!]$ can be partitioned into

$$[\![0, P-1]\!] = \mathcal{I}_{\text{noc}} \cup \mathcal{I}_{\text{mid}} \cup \mathcal{I}_{\text{hi}},$$

where the partitions correspond to the different contention levels. So, by definition, $\mathcal{I}_{\text{noc}} = \{0\}$, and for all $i \in \mathcal{I}_{\text{noc}} \cup \mathcal{I}_{\text{mid}}$, $e(i) = 0$ (see Section 3.3.3.2).

The success period is highly-contended, *i.e.* we have a continuous sequence of *CAS*'s in the success period, if the sum of the execution time of all the *CAS*'s that need to be operated exceeds the critical work. Hence $\mathcal{I}_{\text{hi}} = [\![i_{\text{hi}}, P-1]\!]$, where

$$i_{\text{hi}} = \min\{i \in [\![1, P-1]\!] \mid (i+1) \times cc > cw\}.$$

In addition, as the sequence of *CAS*'s is continuous when the contention is high, the slack time is null when the success period is highly contended, *i.e.*, for all $i \in \mathcal{I}_{\text{hi}}$, $st(i) = 0$, and *a fortiori*, $st(\mathcal{S}_i \to \mathcal{S}_\star) = 0$.

Otherwise, the success period is in medium contention, hence we have $\mathcal{I}_{\text{mid}} = [\![1, i_{\text{hi}} - 1]\!]$. Moreover, if $i \in \mathcal{I}_{\text{mid}}$, $st(i) > 0$, and $e(i) = 0$, because the *CAS*'s synchronized during the previous success period and will not collide any more in the current success period.

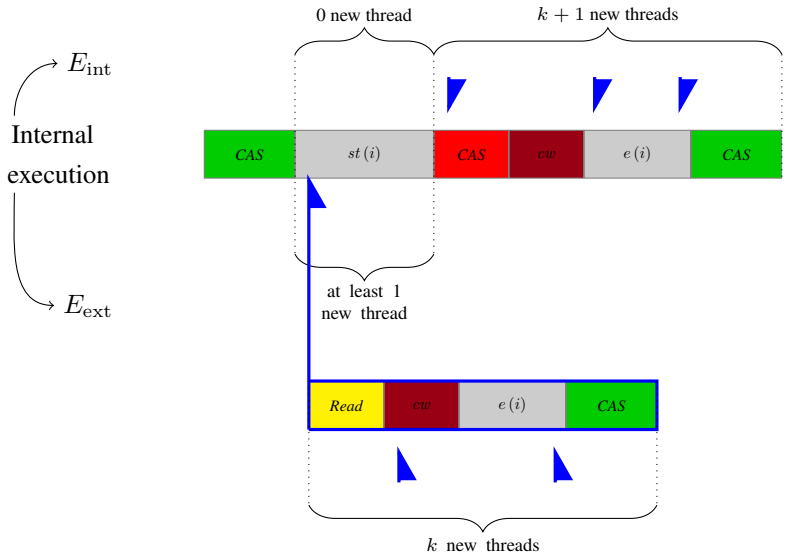


Figure 3.4: Possible executions

3.5.3.1 Transition Matrix

We consider here that the system is in a given state, and we compute the probability that the system will next reach any other state. Without loss of generality, we can choose the origin of time such that the current success period begins at $t = 0$.

Let us first look at the core cases, *i.e.* let $i \in \mathcal{I}_{\text{mid}} \cup \mathcal{I}_{\text{hi}}$ and $k \in [\![0, P - i - 1]\!]$; we assume that the system is currently in state $\mathcal{S}_i$, and we are interested in the probability that the system will switch to $\mathcal{S}_{i+k}$ at the end of the current state. In other words, we want to find the probability that, given that the current success period started when i threads were in the retry loop, the next success period will begin while $i + k$ threads are in the retry loop.

As the successful thread will exit the retry loop at the end of the current success period, there is at least one thread that enters the retry loop during the current success period. Two non-overlapping events can then occur (see Fig-

ure 3.4): either the first thread exiting the parallel section starts within $[0, st\,(i)\,[$, *i.e.* in the slack time of the internal execution, and this event is written E_{ext}, or the first thread entering the retry loop starts after $t = st\,(i)$, and this event is denoted by E_{int}. Therefore, we have $\mathbb{P}\,(\mathcal{S}_i \to \mathcal{S}_{i+k}) = \mathbb{P}\,(E_{\text{ext}}) + \mathbb{P}\,(E_{\text{int}})$.

First note that E_{ext} cannot happen when the success period is highly contended; in this case, the slack time is indeed null, and we conclude $\mathbb{P}\,(E_{\text{ext}}) = 0$. In addition, we have seen in Section 3.5.2 that external threads, *i.e.* threads that are in the parallel section at the beginning of the success period, do not participate to the game of expansion, so they cannot be successful. Under high-contention, E_{int} happens, and the successful *CAS* that ends the success period is operated by an internal thread, *i.e.* a thread that was already in the retry loop when the success period began.

Under medium contention, E_{ext} can occur. In this case, an external thread accesses the data structure before any internal thread does. We have also seen that the expansion is null in medium contention level, thus the external thread will execute its critical work, and especially its *CAS* without being delayed; this implies that the first external thread that accesses the data structure will end the current success period with the end of its *CAS*. If however E_{int} occurs, an internal thread succeeds, but is not necessarily the first thread that accessed the data structure during the success period.

The two possible events are pictured in Figure 3.4, where the blue arrows represent the threads that exit the parallel section. Recall, we aim at computing the probability to start the next success period with $i + k$ threads inside the retry loop. We formalize the idea drawn in the figure by using $X_{[a,b[}$, which is defined as a random variable indicating the number of threads exiting the parallel section during the time interval $[a, b[$. The probability of having E_{int} is then given by

$$\begin{aligned}\mathbb{P}\,(E_{\text{int}}) =& \mathbb{P}\left(X_{[0,st(i)[} = 0 \quad | \quad P_{rl} = i \text{ at } t = 0^+\right) \\ & \times \mathbb{P}\left(X_{[st(i),st(i)+ct(i)[} = k+1 \quad | \quad P_{rl} = i \text{ at } t = st\,(i)^+\right)\end{aligned}$$

Concerning E_{ext}, we know that if $i \in \mathcal{I}_{\text{hi}}$, then $\mathbb{P}\,(E_{\text{ext}}) = 0$. Otherwise, if

we denote by t_3 the starting time of the first thread that exits the parallel section, we obtain

$$\begin{aligned}\mathbb{P}\left(E_{\text{ext}}\right) =& \mathbb{P}\left(X_{[0,st(i)[} > 0 \quad | \quad P_{rl} = i \text{ at } t = 0^+\right) \\ & \times \mathbb{P}\left(X_{[t_3,t_3+rc+cw+cc[} = k \quad | \quad P_{rl} = i+1 \text{ at } t = t_3^+\right)\end{aligned}$$

To simplify the reasoning, and given that the costs of *Read* and *CAS* are approximately the same, we approximate $t_3 + rc + cw + cc$ with $t_3 + cc + cw + cc$, leading to

$$\begin{aligned}\mathbb{P}\left(E_{\text{ext}}\right) =& \mathbb{P}\left(X_{[0,st(i)[} > 0 \quad | \quad P_{rl} = i \text{ at } t = 0^+\right) \\ & \times \mathbb{P}\left(X_{[t_3,t_3+ct(i+1)[} = k \quad | \quad P_{rl} = i+1 \text{ at } t = t_3^+\right)\end{aligned}$$

According to the exponential distribution, given a thread that is in the parallel section at $t = a$, the probability to exit the parallel section within $[a, b[$ is:

$$\int_a^b \lambda e^{-\lambda(t-a)}\, dt = \int_0^{b-a} \lambda e^{-\lambda u}\, du.$$

It is also the probability, given a thread that is in the parallel section at $t = 0$, to exit the retry loop within $[0, b-a[$. This implies:

$$\begin{aligned}\mathbb{P}\left(E_{\text{int}}\right) =& \mathbb{P}\left(X_{[0,st(i)[} = 0 \quad | \quad P_{rl} = i \text{ at } t = 0^+\right) \\ & \times \mathbb{P}\left(X_{[0,ct(i)[} = k+1 \quad | \quad P_{rl} = i \text{ at } t = 0^+\right)\end{aligned}$$

and

$$\begin{aligned}\mathbb{P}\left(E_{\text{ext}}\right) =& \mathbb{P}\left(X_{[0,st(i)[} > 0 \quad | \quad P_{rl} = i \text{ at } t = 0^+\right) \\ & \times \mathbb{P}\left(X_{[0,ct(i)[} = k \quad | \quad P_{rl} = i+1 \text{ at } t = 0^+\right).\end{aligned}$$

To lighten the notations, let us define

$$\begin{cases} a_{i,k} = \mathbb{P}\left(X_{[0,ct(i)[} = k \quad | \quad P_{rl} = i \text{ at } t = 0\right) \\ b_i = \mathbb{P}\left(X_{[0,st(i)[} = 0 \quad | \quad P_{rl} = i \text{ at } t = 0\right). \end{cases} \tag{3.14}$$

In addition, given a thread that is in the parallel section at $t = 0$, the probability to exit the parallel section within $[0, b-a[$ is $\int_0^{b-a} \lambda e^{-\lambda u}\, du$. By counting the number of threads that need to exit the parallel section, we obtain:

$$\begin{cases} a_{i,k} = \binom{P-i}{k} \left(1 - e^{-\lambda ct(i)}\right)^k \left(e^{-\lambda ct(i)}\right)^{P-i-k} \\ b_i = \left(\exp\left(-\lambda st\left(i\right)\right)\right)^{P-i}. \end{cases} \tag{3.15}$$

Altogether, we have that

$$\mathbb{P}\left(\mathcal{S}_i \to \mathcal{S}_{i+k}\right) = b_i \times a_{i,k+1} + (1 - b_i) \times a_{i+1,k}.$$

The situation is slightly different if $k = -1$; in this case, no thread should exit the parallel section during the slack time and no thread should exit during the retry of the first thread that accessed the data structure during the success period neither. This shows that

$$\mathbb{P}\left(\mathcal{S}_i \to \mathcal{S}_{i-1}\right) = b_i \times a_{i,0}.$$

When the success period is not contended, *i.e.* if $i = 0$, the slack time of the execution that ignores external threads can be seen as infinite, hence we can define $b_0 = 0$ (the probability that a thread exits its parallel section during an infinite interval of time is 1). As for the $a_{i,k}$'s, they can be defined in the same way as earlier.

We have obtained the full transition matrix $(M_{i,j})_{(i,j)\in[\![0,P-1]\!]^2}$, which is a triangular matrix, augmented with a subdiagonal:

$$\begin{cases} M_{i,i+k} & = b_i a_{i,k+1} + (1 - b_i) a_{i+1,k} & \text{if } k \in [\![0, P-i-1]\!] \\ M_{i,i-1} & = b_i \times a_{i,0} & \text{if } i > 0 \\ M_{i,j} & = 0 & \text{otherwise} \end{cases}$$

Lemma 14. *M is a right stochastic matrix.*

Proof. First note that, by definition of $a_{i,k}$, for all $i \in [\![0, P-1]\!]$,

$$\sum_{k=0}^{P-i} a_{i,k} = 1.$$

If i threads are indeed inside the retry loop at $t = 0$, then, within $[0, st(i)[$, at least 0 thread, and at most $P - i$ threads (inclusive) will exit their parallel section.

We have first

$$\sum_{j=0}^{P-1} M_{0,j} = \sum_{k=0}^{P-1} a_{0+1,k} = 1.$$

In the same way, for all $i \in [\![1, P-1]\!]$,

$$\begin{aligned}\sum_{j=0}^{P-1} M_{i,j} &= \sum_{k=-1}^{P-1-i} M_{i,i+k} \\ &= b_i \times a_{i,0} + \sum_{k=0}^{P-1-i} b_i a_{i,k+1} + (1-b_i) a_{i+1,k} \\ &= b_i \times \sum_{k=-1}^{P-1-i} a_{i,k+1} + (1-b_i) \sum_{k=0}^{P-1-i} a_{i+1,k}\end{aligned}$$

$$\sum_{j=0}^{P-1} M_{i,j} = 1.$$

□

Lemma 15. *The transition matrix has a unique stationary distribution, which is the unique left eigenvector of the transition matrix with eigenvalue 1 and sum of its elements equal to 1.*

Proof. Note that the Markov chain is irreducible and aperiodic. Let $X \geq P-1$, $i \in [\![0, P-1]\!]$ and $j \in [\![i, P-1]\!]$.

$$\begin{aligned}\mathbb{P}\left(\mathcal{S}_j \to \mathcal{S}_i \text{ in X steps}\right) \geq & \mathbb{P}\left(\mathcal{S}_j \to \mathcal{S}_{j-1} \to \cdots \to \mathcal{S}_i\right) \\ & \times \mathbb{P}\left(\mathcal{S}_i \to \mathcal{S}_i\right)^{X-(j-i)}\end{aligned}$$

$$\mathbb{P}\left(\mathcal{S}_j \to \mathcal{S}_i \text{ in X steps}\right) > 0$$

As

$$\mathbb{P}\left(\mathcal{S}_i \to \mathcal{S}_j \text{ in X steps}\right) \geq \mathbb{P}\left(\mathcal{S}_i \to \mathcal{S}_j\right) > 0,$$

the Markov chain is irreducible. Since $\mathcal{S}_1$ is clearly aperiodic, and the chain is irreducible, the chain is aperiodic as well.

This implies that the Markov chain has a unique stationary distribution, which is the unique left eigenvector of the transition matrix with eigenvalue 1 and sum of its elements equal to 1. □

3.5.3.2 Stationary Distribution

Theorem 7. *Given the transition matrix, the stationary distribution can be found in* $(P+1)P-1$ *operations.*

Proof. As the Markov chain is irreducible, the stationary distribution does not contend any zero. The space of the left eigenvectors with unit eigenvalue is unidimensional; therefore, for any v_0, there exists a vector $v = (v_0\ v_1\ \dots\ v_{P-1})$, such that v spans this space.

Let v_0 a real number; necessarily, v fulfills $v \cdot M = v$, hence for all $i \in [\![0, P-2]\!]$

$$\sum_{k=0}^{i+1} v_k M_{k,i} = v_i,$$

which leads to, for all $i \in [\![0, P-2]\!]$:

$$v_{i+1} = \frac{1}{M_{i+1,i}} \left((1 - M_{i,i}) v_i - \sum_{k=0}^{i-1} v_k M_{k,i} \right).$$

So we obtain the $v_1, \dots, v_{P-1}$ iteratively (we know that $M_{i+1,i} = b_{i+1} \times a_{i+1,0}$, which is not null), with $2 \times i + 1$ operations needed to compute v_{i+1}.

The elements of the stationary distribution should sum to one, so we start from any v_0, compute the whole vector, and then normalize each element by their sum, hence the theorem. □

3.5.3.3 Slack time and Throughput

In order to compute the final throughput, we have to compute the expectation of the slack time, when the system goes from state $\mathcal{S}_i$ to any other state,

that we note $\mathbb{E}\left(st\left(\mathcal{S}_i \to \mathcal{S}_\star\right)\right)$. Also, we will be able to exhibit a vector $s = (s_0, s_1, \ldots, s_{P-1})$ of expected success period, where s_i is the expectation of the execution time of the success period if i threads are in the retry loop when the success period begins:

$$\begin{cases} s_i = \mathbb{E}\left(st\left(\mathcal{S}_i \to \mathcal{S}_\star\right)\right) + cc + cw + e\left(i\right) + cc & \text{if } i \notin \mathcal{I}_{\text{noc}} \\ s_i = \mathbb{E}\left(st\left(\mathcal{S}_i \to \mathcal{S}_\star\right)\right) + rc + cw + cc & \text{otherwise.} \end{cases}$$

Finally, the expected throughput (inverse of the success period) is calculated through

$$T = \frac{1}{v \cdot s},$$

where v is the stationary distribution of the Markov chain.

We know already that if $i \in \mathcal{I}_{\text{hi}}$, then $\mathbb{E}\left(st\left(\mathcal{S}_i \to \mathcal{S}_{i+k}\right)\right) = 0$.

In the other extreme case, *i.e.* if $i \in \mathcal{I}_{\text{noc}}$, we rely on the following lemma.

Lemma 16. *Let an integer n, a real number λ, and n independent random variables $X_1, X_2, \ldots, X_n$, following an exponential distribution of mean λ^{-1}. Let then X be the random variable defined by: $X = \min_{i \in [\![1,n]\!]} X_i$. The expectation of X is:*

$$\mathbb{E}\left(X\right) = \frac{1}{\lambda n}.$$

Proof. We have

$$\begin{aligned} \mathbb{P}\left(X > x\right) &= \mathbb{P}\left(\forall i : X_i > x\right) \\ &= \prod_{i=1}^{n} \mathbb{P}\left(X_i > x\right) \\ &= \left(\int_x^{+\infty} \lambda e^{-\lambda t}\right)^n \\ \mathbb{P}\left(X > x\right) &= e^{-\lambda n x} \end{aligned}$$

Therefore, the probability distribution of X is given by:

$$t \mapsto \lambda n e^{-\lambda n t},$$

and its expectation is computed through

$$\begin{aligned}\mathbb{E}(X) &= \int_0^{+\infty} \lambda n t e^{-\lambda n t}\, dt \\ &= \left[e^{-\lambda n t} t\right]_{+\infty}^{0} + \int_0^{+\infty} e^{-\lambda n t}\, dt \\ &= \left[\frac{1}{\lambda n} e^{-\lambda n t}\right]_{+\infty}^{0} \\ \mathbb{E}(X) &= \frac{1}{\lambda n}\end{aligned}$$

□

This proves that

$$\mathbb{E}\left(st\left(\mathcal{S}_0 \to \mathcal{S}_\star\right)\right) = \frac{pw}{P}.$$

Let now $i \in \mathcal{I}_{\text{mid}}$, and $k \in [\![-1, P-i-1]\!]$; we are interested in $\mathbb{E}\left(st\left(\mathcal{S}_i \to \mathcal{S}_{i+k}\right)\right)$. The slack time is less immediate, and we use the following reasoning. First note that the probability distribution of the first thread exiting the parallel section is given by $t \mapsto \lambda(P-i)e^{-\lambda(P-i)t}$. If this thread comes back during $]0, st(i)[$, the time that passed since the beginning of the success period is the slack time, otherwise, it is $st(i)$.

$$\begin{aligned}&\mathbb{E}\left(st\left(\mathcal{S}_i \to \mathcal{S}_\star\right)\right) \\ &= \int_0^{st(i)} \lambda(P-i)e^{-\lambda(P-i)t} t\, dt + \int_{st(i)}^{+\infty} \lambda(P-i)e^{-\lambda(P-i)t} st(i)\, dt \\ &= \left[e^{-\lambda(P-i)t} t\right]_{st(i)}^{0} + \left[\frac{1}{\lambda(P-i)} e^{-\lambda(P-i)t}\right]_{st(i)}^{0} + st(i)\left[e^{-\lambda(P-i)t}\right]_{+\infty}^{st(i)} \\ &= -st(i)\, e^{-\lambda(P-i)st(i)} + \frac{1 - e^{-\lambda(P-i)st(i)}}{\lambda(P-i)} + st(i)\left(e^{-\lambda(P-i)st(i)}\right)\end{aligned}$$

We conclude that

$$\mathbb{E}\left(st\left(\mathcal{S}_i \to \mathcal{S}_\star\right)\right) = \frac{1 - e^{-\frac{(P-i)st(i)}{pw}}}{P-i} pw.$$

Putting all together, we obtain

$$\begin{cases} \mathbb{E}\left(st\left(\mathcal{S}_i \to \mathcal{S}_\star\right)\right) = \frac{1-e^{-\frac{(P-i)st(i)}{pw}}}{P-i} pw & \text{if } i \in \mathcal{I}_{\text{noc}} \cup \mathcal{I}_{\text{mid}} \\ \mathbb{E}\left(st\left(\mathcal{S}_i \to \mathcal{S}_\star\right)\right) = 0 & \text{if } i \in \mathcal{I}_{\text{hi}}. \end{cases}$$

3.5.3.4 Number of Failed Retries

Another metric to estimate the quality of the model is the number of failed retries per successful retry. We compute it by counting the number of failed retries within the current success period, where a retry is billed to a given success period if its failed *CAS* occurs during this success period. We denote by $\mathbb{E}\left(f_i\right)$ the expected number of failed *CAS* during a success period that begins with i threads, where $i \in [\![0, P-1]\!]$.

If the success period is not contended, *i.e.* if $i \in \mathcal{I}_{\text{noc}}$, no failure will occur since the first *CAS* of the success period will be a success; hence $\mathbb{E}\left(f_i\right) = 0 = i$.

If the success period is mid-contended, *i.e.* if $i \in \mathcal{I}_{\text{mid}}$, every thread that is in the retry loop in the beginning of the success period will execute at least one *CAS* during this success period, and exactly two if the thread is the successful one. We know indeed that, even if a thread exits its parallel section during the slack time, and is then successful, the failed *CAS*'s will occur before the thread entering the retry loop executes its successful *CAS*. As any thread that exits its parallel section during the success period either is successful at its first *CAS*, or does not operate the *CAS* during the success period, we conclude that: $\mathbb{E}\left(f_i\right) = i$.

If the success period is highly contended, *i.e.* if $i \in \mathcal{I}_{\text{hi}}$, then we know that we have an uninterrupted sequence of failed *CAS*'s, from the beginning of the success period to the last ending successful *CAS*. The expected number of failed *CAS*'s is then directly related to the expected duration of the success period. Recalling that the expansion is given in Theorem 6, we obtain:

$$\mathbb{E}\left(f_i\right) = 1 + \frac{cw + e\left(i\right)}{cc}.$$

3.6 Experiments

To validate our analysis results, we use two main types of lock-free algorithms. In the first place, we consider a set of basic algorithms where operations can be completed with a single successful *CAS*. This set of algorithms includes: (i) synthetic designs, that cover the design space of possible lock-free data structures; (ii) several fundamental designs of data structure operations such as lock-free stacks [18] (Pop, Push), queues [7] (Dequeue), counters [19] (Increment, Decrement). As a second step, we consider more advanced lock-free operations that involve helping mechanisms, and show how to use our analysis in this context. Finally, in order to highlight the benefits of the analysis framework, we show how it can be applied to i) determine a beneficial back-off strategy and ii) optimize the memory management scheme used by a data structure, in the context of an application.

We also give insights about the strengths of our two approaches. On the one hand, the constructive approach exhibits better predictions due to the tight estimation of the failing retries. On the other hand, the average-based approach is applicable to a broader spectrum of algorithmic designs as it leaves room to abstract complicated algorithmic designs.

3.6.1 Setting

We have conducted experiments on an Intel ccNUMA workstation system. The system is composed of two sockets equipped with Intel Xeon E5-2687W v2 CPUs with frequency band 1.2-$3.4.\,\mathrm{GHz}$ The physical cores have private L1, L2 caches and they share an L3 cache, which is $25\,\mathrm{MB}$. In a socket, the ring interconnect provides L3 cache accesses and core-to-core communication. Due to the bi-directionality of the ring interconnect, uncontended latencies for intra-socket communication between cores do not show significant variability.Our model assumes uniformity in the *CAS* and *Read* latencies on the shared cache line. Thus, threads are pinned to a single socket to minimize non-uniformity in *Read* and *CAS* latencies. In the experiments, we vary the number of threads between 4 and 8 since the maximum number of threads that can be used in the

experiments are bounded by the number of physical cores that reside in one socket. We show the experimental results with 8 threads.

In all figures, the y-axis shows two metrics: the throughput values (line), *i.e.* number of operations completed per second, and the ratio of failing to successful retries (multiplied by 10^6 for readability and represented by dashed lines), while the mean of the exponentially distributed parallel work pw is represented on the x-axis. The number of failures per success in the average-based approach is computed as $\overline{P_{rl}} - 1$ and is described in Section 3.5.3.4 for the constructive approach.

We have also added a straightforward upper bound as a baseline approach, which is defined as the minimum of $1/(rc + cw + cc)$ (two successful retries cannot overlap) and $P/(pw + rc + cw + cc)$ (a thread can succeed only once in each work loop).

The estimations, that are derived by our average-based approach, constructive approach and the straightforward upper bound, are referred respectively as: *"Average"*, *""Constructive"* and *"Bound"* in the figures. The actual measurements are prefixed with the word *"Real"*.

3.6.2 Basic Data Structures

Here, we consider lock-free operations that can be completed with a single successful *CAS*. and provide predictions using both the average-based and the constructive approach together with the theoretical upper bound.

3.6.2.1 Synthetic Tests

We first evaluate our models using a set of synthetic tests that have been constructed to abstract different possible design patterns of lock-free data structures (value of cw) and different application contexts (value of pw). The critical work is either constant, or follows a Poisson distribution; in Figures 3.5, 3.6, 3.7, its mean value cw is indicated at the top of the graphs. The real measurements that correspond to these cases is referred as *"Real Constant"* and *"Real Poisson"*.

In Figure 3.5, parallel work is instantiated with exponential distribution. A

steep decrease in throughput, as pw gets low, can be observed for the cases with low cw, that mainly originates due to expansion. When cw is high, performance continues to increase when pw decreases, though slightly. The expansion is in-

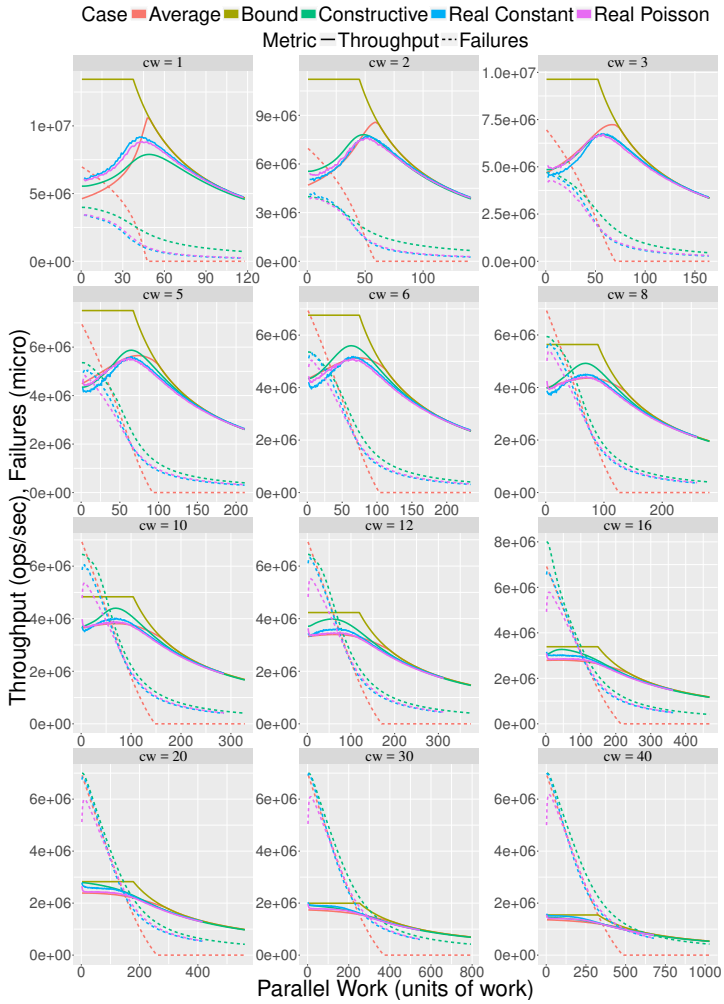


Figure 3.5: Synthetic program with exponentially distributed parallel work

deed low but the slack time, which appears as a more dominant factor, decreases as the number of threads inside the retry loop increases.

When looking into the differences between the constructive and the average-

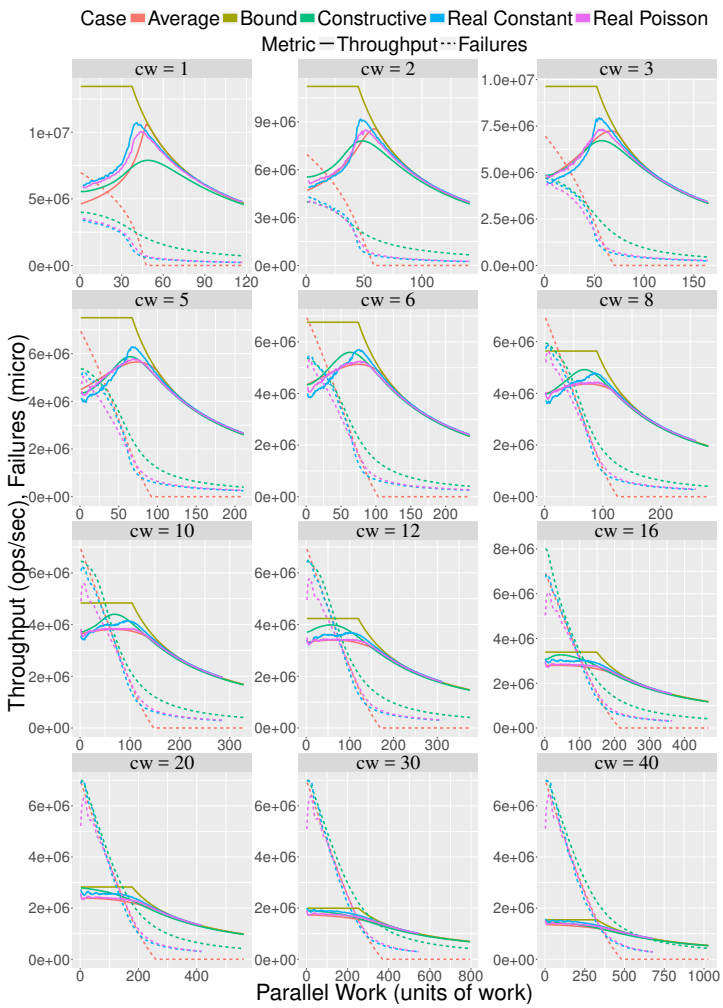


Figure 3.6: Synthetic program with parallel work following Poisson

based approach: the average-based approach estimations come out to be less accurate for mid-contention cases as it only differentiates between contended and non-contended modes. In addition, it fails to capture the failing retries

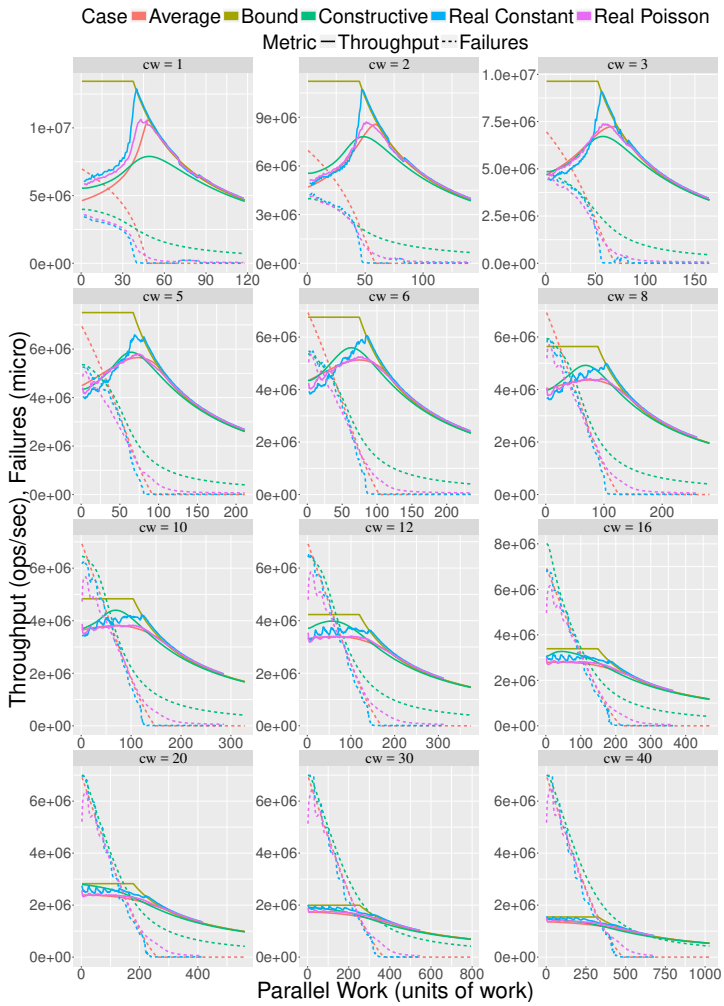


Figure 3.7: Synthetic program with Constant parallel work

when measured throughput starts to deviate from the theoretical upper bound, as pw gets lower. In contrast, the constructive approach provides high accuracy in all metrics for almost every case.

We have also run the same synthetic tests with a parallel work that follows a Poisson distribution (Figure 3.6) or is constant (Figure 3.7), in order to observe the impact of the distribution nature of the parallel work. Compared to the exponential distribution, a better throughput is achieved with a Poisson distribution on the parallel work. The throughput becomes even better with a constant parallel work, since the slack time is minimized due to the synchronization between the threads, as explained in [16]. One can observe this from the difference in start point of failures (with respect to the x-axis) which is also remarkable. Our constructive approach fails to capture the failures in these cases since it is specialized for the exponential distribution.

3.6.2.2 Treiber's Stack

The lock-free stack by Treiber [18] is a fundamental data structure that provides Pop and Push operations. To Pop an element, the top pointer is read and the next pointer of the initial element is obtained. The latter pointer will be the new value of the *CAS* that linearizes the operation. So, accessing the next pointer of the topmost element represents cw as it takes place between the *Read* and the *CAS*. We initialize the stack by pushing elements with or without a stride from a contiguous chunk of memory. By this way, we are able to introduce both costly or not costly cache misses. We also vary the number of elements popped at the same time to obtain different cw; the results, with different cw values are illustrated in Figure 3.8. The results follow a similar trend with the synthetic tests; therefore we skip the discussion to avoid replication.

3.6.3 Towards Advanced Data Structure Designs

Advanced lock-free operations generally require multiple pointer updates that cannot be done with a single *CAS*. One way to design such operations, in a lock-free manner, is to use helping mechanisms: an inconsistency will be fixed

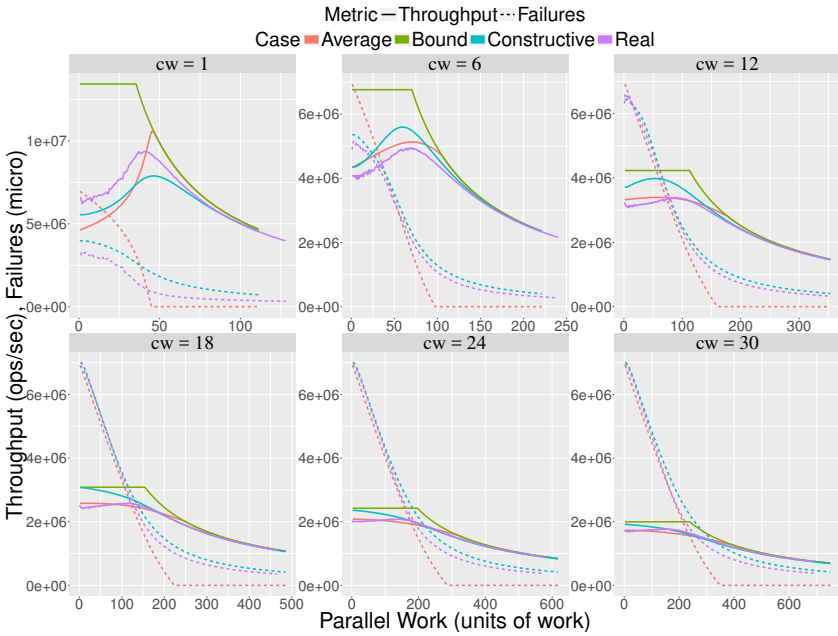


Figure 3.8: Treiber's Stack

eventually by some thread. Here we consider two data structures that apply immediate helping, the queue from [7] and the deque designed in [20]. In the queue experiment (Figure 3.9), we run the Enqueue operation on the queue with and without memory management; in the deque experiment, each thread is dedicated to an end of the deque (equally distributed), while we vary the proportion of push operations (colors in Figure 3.10).

Here, we consider data structures that apply immediate helping, where threads help for the completion of a recently linearized operation until the data structure comes into a stable state in which a new operation can be linearized. The crucial observation is that the data structure goes through multiple stages in a round robin fashion. The first stage is the one where the operation is linearized. The remaining ones are the stages in which other threads, that execute another

operation, might help for the completion of the linearized operation, before attempting to linearize their own operations. Thus, the success period (ignoring the slack time) can be seen as the sum of the execution time of these stages, each ending with a *CAS* that updates a pointer. The *CAS* in the first stage might be expanded by the threads that are competing for the linearization of their operation, and consequent *CAS*'s might be expanded by the helper threads, which are still trying to help an already completed operation. Also, there might be slack time before the start of the first stage as the other stages will start immediately due to the thread that has completed the previous stage.

Although it is hard to stochastically reconstruct the executions with Markov chains, our average-based approach provides the flexibility required to estimate the performance by plugging the expected success period, given the number of threads inside the retry loop, into the Little's Law. As the impacting factors are similar, we estimate the success period in the same vein as in Section 3.4; with a minor adaptation of the expansion formula and by slightly adapting the slack time estimation based on the same arguments.

3.6.3.1 Expected Expansion for the Advanced Data Structures

Consider an operation such that, the success period (ignoring the slack time) is composed of S stages (denoted by $Stage_1, \ldots, Stage_S$) where each stage represents a step towards the completion of the operation. Let CAS_i denote the *CAS* operation at the end of the $Stage_i$. From a system-wide perspective, $\{CAS_1, \ldots, CAS_S\}$ is the set of *CAS*'s that have to be successfully and consecutively executed to complete an operation, assuming all threads are executing the same operation. This design enforces that CAS_i can be successful only if the last successful *CAS* is a CAS_{i-1}. And, CAS_1 can be successful only if the last successful *CAS* is a CAS_S. In other words, another operation can not linearize before the completion of the linearized but incomplete operation.

Now, let e_i denote the expected expansion of CAS_i. If the data structure is in the stable state (*i.e.* is in $Stage_1$, where a new operation can be linearized), then we have to consider the probability, for all threads except one, to expand the successful CAS_1 which linearizes the operation. After the linearization, this

operation will be completed in the remaining stages where again the successful *CAS*'s at the end of the stages are subject to the same expansion possibility by the threads in the retry loop, as they might be still trying to help for the completion of the previously completed operation.

Similar to the [16], we assume that any thread that is in the retry loop with probability h, can launch CAS_i that might expand the successful CAS_i. We consider, the starting point of a failing CAS_i is a random variable which is distributed uniformly within the retry loop, which is composed of expanded stages of the operation. This is because an obsolete thread can launch a CAS_i, regardless of the stage in which the data structure is in (equally, regardless of the last successful *CAS*). Due to the uniformity assumption, the expansion for the successful *CAS*'s in all stages, would be equal. Similar to the [16], we estimate the expansion e_i by considering the impact of a thread that is added to the retry loop. Let the cost function $delay_i$ provide the amount of delay that the additional thread introduces, depending on the point where the starting point of its CAS_i hits. By using these cost functions, we can formulate the total expansion increase that each new thread introduces and derive the differential equation below to calculate the expected total expansion in a success period, where $\overline{e}\left(\overline{P_{rl}}\right) = \sum_{i=1}^{S} \overline{e_i}\left(\overline{P_{rl}}\right)$. Note that, we assume that the expansion starts as soon as strictly more than 1 thread are in the retry loop, in expectation.

Lemma 17. *The expansion of a CAS operation is the solution of the following system of equations, where* $rlw = \sum_{i=1}^{S} rlw_i = \sum_{i=1}^{S}(rc_i + cw_i + cc_i)$*:*

$$\begin{cases} \overline{e}'\left(\overline{P_{rl}}\right) &= cc \times \dfrac{S \times \frac{cc}{2} + \overline{e}\left(\overline{P_{rl}}\right)}{rlw + \overline{e}\left(\overline{P_{rl}}\right)} \\ \overline{e}\left(P_{rl}^{(0)}\right) &= 0 \end{cases}, \text{ where } P_{rl}^{(0)} \text{ is the point that expansion begins.}$$

Proof. We compute $\overline{e}\left(\overline{P_{rl}} + h\right)$, where $h \leq 1$, by assuming that there are already $\overline{P_{rl}}$ threads in the retry loop, and that a new thread attempts to *CAS* during the retry, within a probability h. For simplicity, we denote $a_j^i = (\sum_{j=1}^{i-1} rlw_j + e_j(\overline{P_{rl}})) + rc_i + cw_i$.

$$
\begin{aligned}
&\overline{e}\left(\overline{P_{rl}}+h\right)\\
&=\overline{e}\left(\overline{P_{rl}}\right)+h\times\sum_{i=1}^{S}\int_{0}^{rlw^{(+)}}\frac{delay_i\left(t_i\right)}{rlw^{(+)}}\,dt_i\\
&=\overline{e}\left(\overline{P_{rl}}\right)+h\times\sum_{i=1}^{S}\Big(\int_{0}^{a_j^i-cc}\frac{delay_i\left(t_i\right)}{rlw^{(+)}}\,dt_i+\int_{a_j^i-cc}^{a_j^i}\frac{delay_i\left(t_i\right)}{rlw^{(+)}}\,dt_i\\
&+\int_{a_j^i}^{a_j^i+\overline{e_i}\left(\overline{P_{rl}}\right)}\frac{delay_i\left(t_i\right)}{rlw^{(+)}}\,dt_i+\int_{a_j^i+\overline{e_i}\left(\overline{P_{rl}}\right)}^{rlw^{(+)}}\frac{delay_i\left(t_i\right)}{rlw^{(+)}}\,dt_i\Big)\\
&=\overline{e}\left(\overline{P_{rl}}\right)+h\times\sum_{i=1}^{S}\Big(\int_{a_j^i-cc}^{a_j^i}\frac{t_i}{rlw^{(+)}}\,dt_i+\int_{a_j^i}^{a_j^i+\overline{e_i}\left(\overline{P_{rl}}\right)}\frac{cc}{rlw^{(+)}}\,dt_i\Big)\\
&=\overline{e}\left(\overline{P_{rl}}\right)+h\times\frac{\left(\sum_{i=1}^{S}\frac{cc^2}{2}\right)+\overline{e}\left(\overline{P_{rl}}\right)\times cc}{rlw^{(+)}}
\end{aligned}
$$

This leads to

$$
\frac{\overline{e}\left(P_{rl}+h\right)-\overline{e}\left(\overline{P_{rl}}\right)}{h}=\frac{S\times\frac{cc^2}{2}+\overline{e}\left(\overline{P_{rl}}\right)\times cc}{rlw^{(+)}}.
$$

When making h tend to 0, we finally obtain

$$
\overline{e}'\left(\overline{P_{rl}}\right)=cc\times\frac{S\times\frac{cc}{2}+\overline{e}\left(\overline{P_{rl}}\right)}{rlw+\overline{e}\left(\overline{P_{rl}}\right)}. \qquad \square
$$

In addition, if a set S_k of *CAS*'s are operating on the same variable var_k, then $CAS_i \in S_k$ can be expanded by the $CAS_j \in S_k$. In this case, we can obtain $\overline{e_k}\left(\overline{P_{rl}}\right)$ by using the reasoning above. The calculation simply ends up as follows: Consider the problem as if no *CAS* shares a variable and denote expansion in $Stage_i$ with $\overline{e_i}\left(\overline{P_{rl}}\right)^{(old)}$. Then, $\overline{e_k}\left(\overline{P_{rl}}\right)=\sum_{CAS_i\in S_k}\overline{e_i}\left(\overline{P_{rl}}\right)^{(old)}$.

3.6.3.2 Expected Slack Time for the Advanced Data Structures

We assume here the slack time can only occur after the completion of an operation (*i.e.* before stage 1), as the other stages are expected to start immediately

due to the thread that completes the previous stage. Similar to Section 3.4.1.2, we consider that, at any time, the threads that are running the retry loop have the same probability to be anywhere in their current retry. Thus, a thread can be in any stage just after the successful CAS that completes the operation. So, we need to consider the thread which is closest to the end of its current stage when the operation is completed. We denote the execution time of the expanded retry loop with $rlw^{(+)}$ and the number of stages with S. For a thread executing $Stage_i$ when the operation completes, the time before accessing the data structure is then uniformly distributed between 0 and $rlw_i^{(+)}$.

Here, we take another assumption and consider all stages can be completed in the same amount of time (*i.e.* for all (i, j) in $\{1, \ldots, S\}^2$, $rlw_i^{(+)} = rlw_j^{(+)} = rlw^{(+)}/S$). This assumption does not diverge much from the reality and provides a reasonable approximation. With these assumption and using Lemma 12, we conclude that:

$$\overline{st}\left(\overline{P_{rl}}\right) = \frac{rlw^{(+)}}{S \times (\overline{P_{rl}} + 1)}. \tag{3.16}$$

3.6.3.3 Enqueue on Michael-Scott Queue

As a first step, we consider the Enqueue operation of the MS queue to validate our approach. This operation requires two pointer updates leading to two stages, each ending with a *CAS*. The first stage, that linearizes the operation, updates the next pointer of the last element to the newly enqueued element. In the next and last stage, the queue's head pointer is updated to point to the recently enqueued element, which could be done by a helping thread, that brings the data structure into a stable state. Here, we determine the cw by subtracting the rc and cc from the non-contended cost of Enqueue operation.

We estimate the expansion in the success period as described above and throughput as explained in Section 3.4. The results for the Enqueue experiments where all threads execute Enqueue are presented in Figure 3.9. Without memory management, the operation provides better performance because cw value is smaller in this case. With the injection of memory management in-

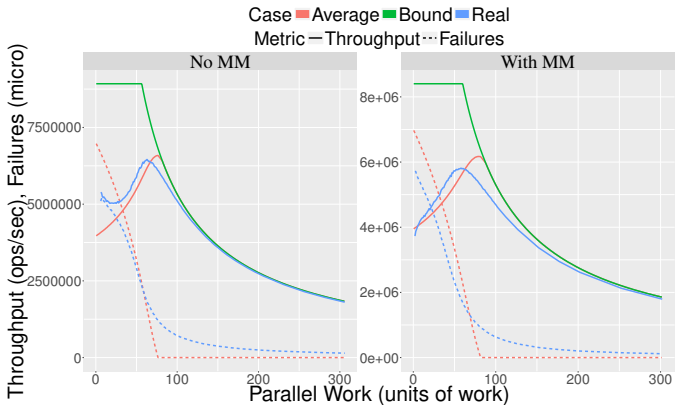


Figure 3.9: Enqueue on MS Queue

structions, retry loop size grows and performance decreases. Our average-based approach manages to capture the performance in both cases with satisfactory precision.

3.6.3.4 Deque

We consider the deque designed in [20]. PushLeft and PushRight (resp. PopLeft and PopRight) operations are exactly the same, except that they operate on the different ends of the deque. The status flags, which depict the state of the deque, and the pointers to the leftmost element and the rightmost element are together kept in a single double-word variable, so-called *Anchor*, which could be modified by a double-word *CAS* atomically.

A PopLeft operation linearizes and even completes in one stage that ends with a double-word *CAS* that just sets the left pointer of the anchor to the second element from left.

A PushLeft operation takes three stages to complete. In the first stage, the operation is linearized by setting the left pointer of the *Anchor* to the new element and at the same time changing the status flags to "left unstable", to indicate the status of the incomplete but linearized PushLeft operation. In the

second stage, the left pointer of the leftmost element is redirected to the recently pushed element. In the third stage, a *CAS* is executed on *Anchor* to bring the deque status flags into "stable state". Every operation can help an incomplete PushLeft or PushRight until the deque comes into the stable state; in this state, the other operations can attempt to linearize anew.

As noticed, the first and the third stage execute a *CAS* on the same variable (*Anchor*) so it is possible to delay the third stage of the success period by executing a *CAS* in the first stage. This implies that the expansion in stage one should also be considered when the delay in the third stage is considered, and the other way around. This can be done by summing expansion estimates of the stages that run the *CAS* on the same variable and using this expansion value in all these stages. Again, it just requires simple modifications in the expansion formula by keeping assumptions unchanged.

We first run pop-only and push-only experiments where dedicated threads operate on both ends of the deque, in a half-half manner. We provide predictions by plugging the slightly modified expansion estimate, as explained above, into the average-based approach. Then, we take one step further and mix the operations, assigning the threads inequally among push and pop operations. And, we obtain estimates for them by simply taking the weighted average (depending on the number of threads running each operation) of the success period of pop-only and push-only experiments, with the corresponding pw value.

In Figure 3.10, results are illustrated; they are satisfactory for the push-only and pop-only cases. For the mixed-case experiments, the results are mixed: our analysis follows the trend and becomes less accurate when the pw gets lower, as experimental curves tend toward push-only success period. This, presumably, happens because the first stage of a PushLeft (or PushRight) operation is shorter than the first stage of a PopLeft (or PopRight) operation. This brings indeed an advantage to push operations, under contention: they have higher chances to linearize before pop operations after the data structure comes into the stable state. It also provides an interesting observation which highlights the lock-free nature of operations: it is improbable to complete a pop operation if numerous threads try to push, due to the difference of work inside the first stage

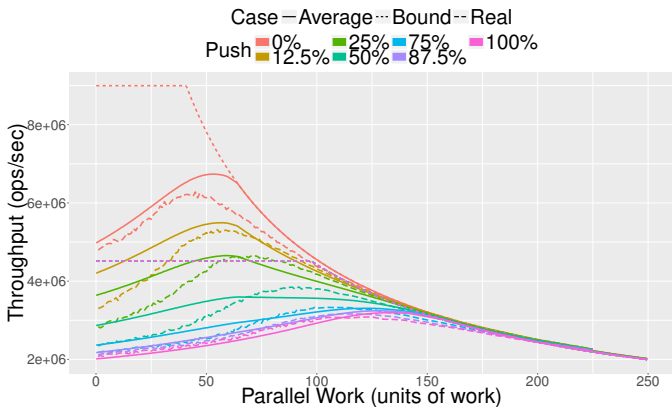


Figure 3.10: Operations on deque

of their retry loop.

3.6.4 Applications

3.6.4.1 Back-off Optimizations

When the parallel work is known, we can deduce from our analysis a simple and efficient back-off strategy: as we are able to estimate the value for which the throughput is maximum, we just have to back-off for the time difference between the peak pw and the actual pw. In Figure 3.12, we compare, on a synthetic workload, this constant back-off strategy against widely known strategies, namely exponential and linear, where the back-off amount increases exponentially or linearly after each failing retry loop starting from a $115\,\mathrm{cycles}$ step size. In Figure 3.11, we apply our constant back-off on a Delaunay triangulation application [11], provided with several workloads. The application uses a stack in two phases, whose first phase pushes elements on top of the stack without delay. We are able to estimate a corresponding back-off time, and we plot the results by normalizing the execution time of our back-offed implementation with the execution time of the initial implementation.

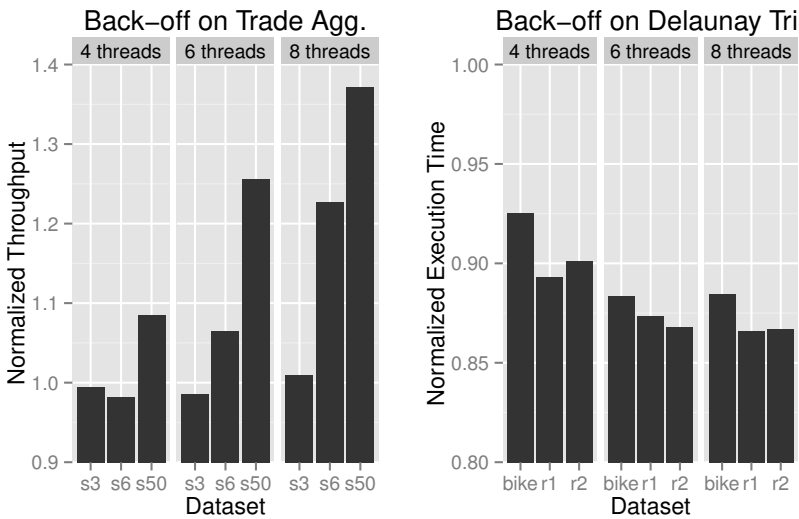


Figure 3.11: Performance impact of our back-off tunings

A measure or an estimate of pw is not always available (and could change over time, see next section), therefore we propose also an adaptive strategy: we incorporate in the data structure a monitoring routine that tracks the number of failed retries, employing a sliding window. As our analysis computes an estimate of the number of failed retries as a function of pw, we are able to estimate the current pw, and hence the corresponding back-off time like previously.

We test our adaptive back-off mechanism on a workload originated from [12], where global operators of exchanges for financial markets gather data of trades with a microsecond accuracy. We assume that the data comes from several streams, each of them being associated with a thread. All threads enqueue the elements that they receive in a concurrent queue, so that they can be later aggregated. We extract from the original data a trade stream distribution that we use to generate similar streams that reach the same thread; varying the number of streams to the same thread leads to different workloads. The results, represented as the normalized throughput (compared to the initial throughput) of trades that

are enqueued when the adaptive back-off is used, are plotted in Figure 3.11. For any number of threads, the queue is not contended on workload s3, hence our improvement is either small or slightly negative. On the contrary, the workload s50 contends the queue and we achieve very significant improvement.

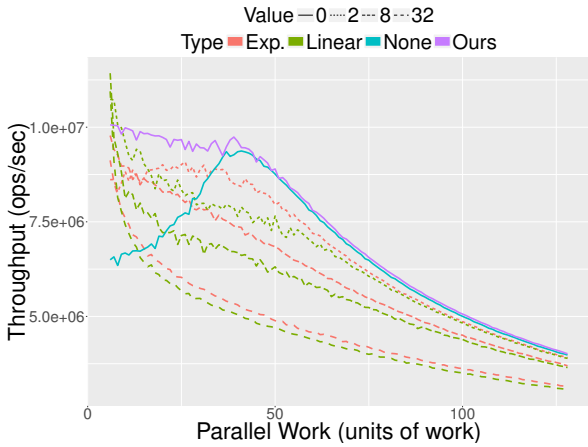


Figure 3.12: Back-off Tuning on Treiber's Stack

3.6.4.2 Memory Management Optimization

Memory Management (MM) is an inseparable part of dynamic concurrent data structures. In contrary to lock-based implementations, a node that has been *removed* from a lock-free data structure can still be accessed by other threads, *e.g.* if they have been delayed. Collective decisions are thus required in order to *reclaim* a node in a safe manner. A well-known solution to deal with this problem is the hazard pointers technique [21].

A traditional design to implement this technique works as follows. Each thread $\mathcal{T}_i$, maintains two lists of nodes: $\mathcal{N}_i$ contains the nodes that $\mathcal{T}_i$ is currently accessing, and $\mathcal{D}_i$ stores the nodes that have been removed from the data structure by $\mathcal{T}_i$. Once a threshold on the size of $\mathcal{D}_i$ is reached, $\mathcal{T}_i$ calls a routine that: (i) collects the nodes that are accessed by any other thread, *i.e.* $\mathcal{N}_j$

for $j \neq i$ (collection phase), and (ii) for each element in $\mathcal{D}_i$, checks whether someone is accessing the element, *i.e.* whether it belongs to $\cup_{j \neq i} \mathcal{N}_j$, and if not, reclaims it (reclamation phase).

The primary goal of our adaptive MM scheme is to distribute this extra-work (harmful under low-contention) in a way that the loss in performance is largely leveraged, knowing that additional work can be an advantage under high-contention (see the previous section). The optimization is based on two main modifications. First, the granularity has to be finer, since the additional quantum that the back-off mechanism uses, has to be rather small (hundreds of cycles for a queue). Second, we need to track the contention level on the data structure in order to be able to inject the work at a proper execution point. Then, the memory management execution can be delayed under low-contention and take place under high-contention in the right amount to obtain the peak performance.

Fine-grain Memory Management Scheme: We divide the routine (and further the phases) of the traditional MM mechanism into quanta (equally-sized chunks).One quantum of the collection phase is the collection of the list of one thread, while three nodes are reclaimed during one quantum of the reclamation phase. The traditional MM scheme was parameterized by a threshold based on the number of the removed nodes; the fine-grain MM scheme is parameterized by the number of quanta that are executed at each call.

We apply different MM schemes on the Dequeue operation of the Michael-Scott queue, and plot the results in Figure 3.13. We initialize the queue with enough elements. Threads execute Dequeue, which returns an element, then call the MM scheme. On the left side, we compare a pure queue (without MM), a queue with the traditional MM (complete reclamation once in a while) and a queue with fine-grain MM (according to the numbers of quanta that are executed at each call, given by *"Parameter"* in the legend which is effective only for fine-grain MM). Note that the performance of the traditional MM is also subject to the tuning of the threshold parameter. We have tested and kept only the best parameter on the studied domain. First, unsurprisingly, we can observe that the pure queue outperforms the others as its cw is lower (no need to maintain the

list of nodes that a thread is accessing). Second, as the fine-grain MM is called after each completed Dequeue, adding a constant work, the MM can be seen as a part of the parallel work. We highlight this idea on the second experiment (on the right side). We first measure the work done in a quantum. It follows that, for each value of the granularity parameter, we are able to estimate the effective parallel work as the sum of the initial pw and the work added by the fine-grain MM. Finally, we run the queue with the fine-grain MM, and plot the measured throughput, according to the effective parallel work, together with our two approaches instantiated with the effective pw. The graph shows the validity of the model estimations for all values of the granularity parameter.

Adaptive Memory Management Scheme: We build the adaptive MM scheme on top of the fine-grain MM mechanism by adding a monitoring routine that tracks the number of failed retry loops, employing a sliding windows. Given a

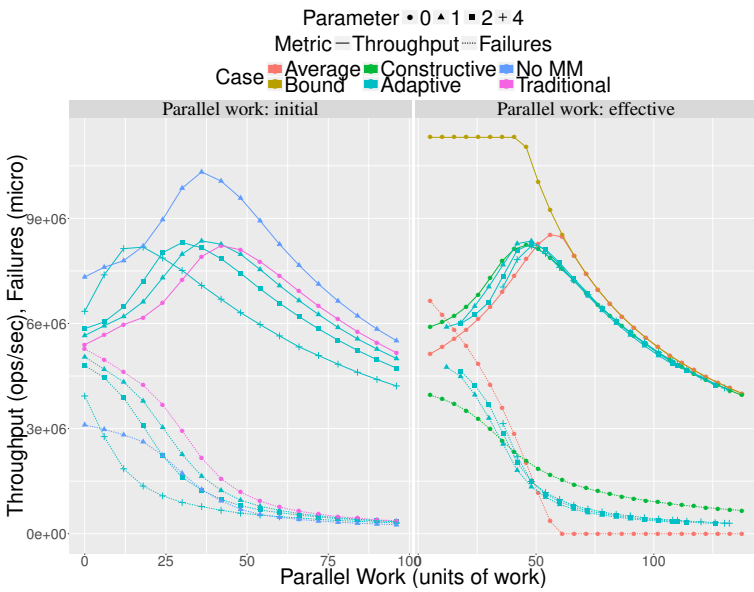


Figure 3.13: Performance of memory management mechanisms

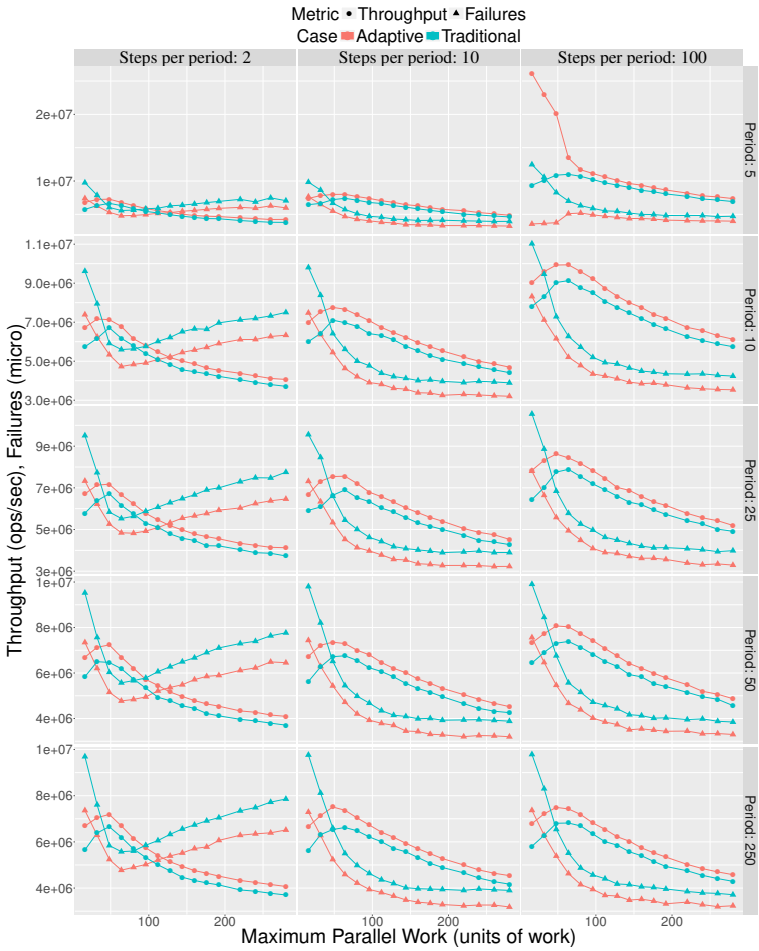


Figure 3.14: Adaptive MM with varying mean pw

granularity parameter and a number of failed retry loops, we are able to estimate the parallel work and the throughput, hence we can decide a change in the granularity parameter to reach the peak performance. Note that one can avoid memory explosion by specifying a threshold like the traditional implementation

in case the application provides a durable low contention; in the worst case, it performs like the traditional MM.

Numerous scientific applications are built upon a pattern of alternating phases, that are communication- or computation-intensive. If the application involves data structures, it is expected that the rate of the modifications to the data structures is high in the data-oriented phases, and conversely. These phases could be clearly separated, but the application can also move gradually between phases. The rate of modification to a data structure will anyway oscillate periodically between two extreme values. We place ourselves in this context, and evaluate the two MMs accordingly. The parallel work still follows an exponential distribution of mean pw, but pw varies in a sinusoidal manner with time, in order to emulate the numerical phases. More precisely, pw is a step approximation of a sine function. Thus, two additional parameters rule the experiment: the period of the oscillating function represents the length of the phases, and the number of steps within a period depicts how continuous are the phase changes.

In Figure 3.14, we compare our approach with the traditional implementation for different periods of the sine function, on the Dequeue of the Michael-Scott queue [7]. The adaptive MM, that relies on the analysis presented in this paper, outperforms the traditional MM because it provides an advantage both under low contention due to the costless (since delayed) invocation of the MM and under high contention due to the back-off effect.

3.7 Conclusion

In this paper we have presented two analyses for calculating the performance of lock-free data structures in dynamic environments. The first analysis has its roots in queuing theory, and gives the flexibility to cover a large spectrum of configurations. The second analysis makes use of Markov chains to exhibit a stochastic execution; it gives better results, but it is restricted to simpler data structures and exponentially distributed parallel work. We have evaluated the quality of the prediction on basic data structures like stacks, as well as more advanced data structures like optimized queues and deques. Our results can

be directly used by algorithmicians to gain a better understanding of the performance behavior of different designs, and by experimentalists to rank implementations within a fair framework. We have also shown how to use our results to tune applications using lock-free codes. These tuning methods include: (i) the calculation of simple and efficient back-off strategies whose applicability is illustrated in application contexts; (ii) a new adaptative memory management mechanism that acclimates to a changing environment.

The main differences between the data structures of this paper and linked lists, skip lists and trees occur when the size of the data structure grows. With large sizes, the performance is dominated by the traversal cost that is ruled by the cache parameters. The reduction in the size of the data structure decreases the traversal cost which in turn increases the probability of encountering an on-going *CAS* operation that delays the threads which traverse the link. The expansion, which can additionally be supported unfavorably by helping mechanisms, appears then as the main performance degrading factor. While the analysis becomes easier for high degrees of parallelism (large data structure size), being able to describe the behavior of lock-free data structures as the degree of parallelism changes constitutes the main challenge of our future work.

Bibliography

[1] "Intel's threading building blocks framework," `https://www.threadingbuildingblocks.org/`, Accessed: 2016-01-20.

[2] "Java concurrency package," `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html`, Accessed: 2016-01-20.

[3] "Microsoft .net framework," `http://www.microsoft.com/net`, Accessed: 2016-01-20.

[4] Hagit Attiya and Arie Fouren, "Algorithms adapting to point contention," *Journal of the ACM (JACM)*, vol. 50, no. 4, pp. 444–468, 2003.

[5] Yehuda Afek, Gideon Stupp, and Dan Touitou, "Long lived adaptive splitter and applications," *Journal of Distributed Computing*, vol. 15, no. 2, pp. 67–86, 2002.

[6] Hagit Attiya, Rachid Guerraoui, and Petr Kouznetsov, "Computing with reads and writes in the absence of step contention," in *Proceedings of the International Symposium on Distributed Computing (DISC)*. 2005, pp. 122–136, Springer.

[7] Maged M. Michael and Michael L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the ACM Symposium on Principles of Distributed Computing (PoDC)*. 1996, pp. 267–275, ACM.

[8] Danny Hendler, Nir Shavit, and Lena Yerushalmi, "A scalable lock-free stack algorithm," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 70, no. 1, pp. 1–12, 2010.

[9] J. D. Valois, "Implementing Lock-Free Queues," in *Proceedings of International Conference on Parallel and Distributed Systems (ICPADS)*, December 1994, pp. 64–69.

[10] Maurice Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 1, pp. 124–149, 1991.

[11] Tanmay Gangwani, Adam Morrison, and Josep Torrellas, "CASPAR: breaking serialization in lock-free multicore synchronization," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2016, pp. 789–804, ACM.

[12] "Daily trades from 2015-08-05," `http://www.nyxdata.com/Data-Products/Daily-TAQ#155`, Accessed: 2016-05-05.

[13] Xiao Yu, Zhengyu He, and Bo Hong, "A queuing model-based approach for the analysis of transactional memory systems," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 6, pp. 808–825, 2013.

[14] Samy Al-Bahra, "Nonblocking algorithms and scalable multicore programming," *Communications of the ACM*, vol. 56, no. 7, pp. 50–61, 2013.

[15] Dan Alistarh, Keren Censor-Hillel, and Nir Shavit, "Are lock-free concurrent algorithms practically wait-free?," in *Proceedings of the ACM Symposium on Theory of Computing (STOC)*. 2014, pp. 714–723, ACM.

[16] Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas, "Analyzing the performance of lock-free data structures: A conflict-based model," in *Proceedings of the International Symposium on Distributed Computing (DISC)*. 2015, pp. 341–355, Springer.

[17] John D. C. Little, "A proof for the queuing formula: L= λ w," *Operations research*, vol. 9, no. 3, pp. 383–387, 1961.

[18] R. Kent Treiber, *Systems programming: Coping with parallelism*, International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[19] Dave Dice, Yossi Lev, and Mark Moir, "Scalable statistics counters," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2013, pp. 43–52, ACM.

[20] Maged M. Michael, "Cas-based lock-free algorithm for shared deques," in *Euro-Par COnference*. 2003, pp. 651–660, Springer.

[21] Maged M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 15, no. 6, pp. 491–504, 2004.

RESULT III

Aras Atalar, Anders Gidenstam, Paul Renaud-Goud and Philippas Tsigas
Modeling Energy Consumption of Lock-Free Queue Implementations
In the Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2015)
pages 229-238, IEEE Press 2015.

4

RESULT III - Modeling Energy Consumption of Lock-Free Queue Implementations

Abstract

This paper considers the problem of modeling the energy behavior of lock-free concurrent queue data structures. Our main contribution is a way to model the energy behavior of lock-free queue implementations and parallel applications that use them. Focusing on steady state behavior we decompose energy behavior into throughput and power dissipation which can be modeled separately and later recombined into several useful metrics, such as energy per operation. Based on our models, instantiated from synthetic benchmark data, and using

only a small amount of additional application specific information, energy and throughput predictions can be made for parallel applications that use the respective data structure implementation. To model throughput we propose a generic model for lock-free queue throughput behavior, based on a combination of the dequeuers' throughput and enqueuers' throughput. To model power dissipation we commonly split the contributions from the various computer components into static, activation and dynamic parts, where only the dynamic part depends on the actual instructions being executed. To instantiate the models a synthetic benchmark explores each queue implementation over the dimensions of processor frequency and number of threads. Finally, we show how to make predictions of application throughput and power dissipation for a parallel application using a lock-free queue requiring only a limited amount of information about the application work done between queue operations. Our case study on a Mandelbrot application shows convincing prediction results.

4.1 Introduction

Lock-free implementations of data structures is a scalable approach for designing concurrent data structures. Lock-free data structures offer high concurrency and immunity to deadlocks and convoying, in contrast to their blocking counterparts. Concurrent FIFO queue data structures are fundamental data structures that are key components in applications, algorithms, run-time and operating systems. The producer/consumer pattern, *e.g.*, is a common approach to parallelizing applications where threads act as either producers or consumers and synchronize and stream data items between them using a shared collection. A concurrent queue, *a.k.a.* shared "first-in, first-out" or FIFO buffer, is a shared collection of elements which supports at least the basic operations Enqueue (adds an element) and Dequeue (removes the oldest element). Dequeue returns the element removed or, if the queue is empty, NULL. A large number of lock-free (and wait-free) queue implementations have appeared in the literature, *e.g.* [1–6] being some of the most influential or most efficient results. Each implementation of a lock-free queue has obviously its strong and weak

points so the impact on performance and energy when choosing one particular implementation for any given situation may not be obvious.

As the number of known implementations of lock-free concurrent queues is growing, it is of great interest to describe a framework within which the different implementations can be ranked, according to the parameters that characterize the situation. A brute force approach could achieve this by running the implementations on hand on the whole domain of study, gathering and comparing measurements. This would yield high accuracy, but at a tremendous cost, since the domain is likely to be large. Additionally, it would only bring a limited understanding on the phenomena that drive the behavior of the queue implementations. Therefore, we propose generic models for predicting the behavior of lock-free queues under steady state usage. The models are instantiated for the queue implementations and machine on hand using empirical data from a limited number of points in the domain.

The implementations can be ranked according to a plethora of metrics. Traditionally, performance in terms of throughput has been the main metric. Furthermore, the notion of energy efficiency has now extended into every nook and cranny of Information Technology, at any scale, from the Exascale machines that need huge improvements in terms of power dissipation to be feasible [7], to the small electronic devices where the battery lifetime is a critical issue.

We decompose the energy behavior of queues, and subsequently applications, into two components: (i) throughput and (ii) power dissipation. We model these components separately. The predicted throughput and power dissipation can be recombined into the energy-efficiency metric energy per queue operation, which is the ratio between power dissipation and queue throughput. When modeling an application, this metric can be extended to energy per unit of application work. Further, plotting energy per operation or unit of work according to throughput allows exploration of the Pareto-optimal frontier of the energy−performance bi-criteria optimization problem for the queues or the application.

Lock-free queue data structures generally offer twofold parallelism: enqueuers and dequeuers modify only their respective ends of the queue, and

compete mostly with operations of the same kind. Nonetheless, when the queue is close to empty, both ends point to the same part of the queue, then enqueue and dequeue operations have to be synchronized, and every operation impacts the behavior of any other.

Concerning the queue as a whole, a successful event can be seen as the dequeue of a non-NULL item, since this event implies that the item has been enqueued and dequeued. Also, the throughput of the queue is naturally defined as the number of such events per unit of time, which is a meaningful performance criterion for queues.

In this work, we focus on queues that are in a steady state, *i.e.* such that the rate of each operation attempt is constant. Then, the throughput $\mathcal{T}$ of the queue is the minimum between the throughput of all dequeues $\mathcal{T}_d$, even those returning NULL, and throughput of enqueues $\mathcal{T}_e$. Indeed, if $\mathcal{T}_e > \mathcal{T}_d$, then the queue grows and the throughput is determined by the dequeuers, which cannot obtain any NULL items; and if $\mathcal{T}_e \leq \mathcal{T}_d$, then the queue is mostly empty and NULL items are dequeued, but the throughput is determined by the enqueuers.

Despite this decomposition, enqueuers' and dequeuers' throughput are still correlated when the queue is mostly empty. In addition, the interactions between them are rather asymmetric, as in broad terms, an enqueue can be delayed by any concurrent dequeue, while for a dequeue, concurrent enqueues will cease to disturb it as they move away from the dequeue end.

Based on these facts, we decorrelate the throughput into several uncorrelated and basic throughputs, and reconstitute the main throughput by combining them. Among the advantages of this process, we earn a better understanding of the performance (as the basic throughputs are meaningful), and we reduce the number of measurements needed to instantiate the model on the whole domain of study.

The domain of study that we envision here can be viewed as the Cartesian product of four sets: (i) number of threads accessing the queue, (ii) CPU frequencies, (iii) a range of dequeue access rates, (iv) a range of enqueue access rate. The cardinality of the first two sets is at most a few tens, while the last two are continuous sets that are not even bounded. In this paper, thanks to the

removal of the dependencies between throughputs, we are able to instantiate the model with only a few data points, while the model covers the whole intervals.

Finally, this decomposition also eases the study of power dissipation, where we reuse the same ideas as in the throughput estimation part.

The rest of the paper is organized as follows. Section 4.2 discusses related work. Section 4.3 introduces our modeling framework for lock-free concurrent queues. Section 4.4 describes how the throughput of lock-free concurrent queues is modeled, while Section 4.5 describes how the power dissipation is modeled. In Section 4.6 we develop a method to model parallel applications using the queue models and apply it to an application for computing the Mandelbrot set. Finally, Section 4.7 concludes this paper.

4.2 Related work

Hunt *et al.* [8] measured the performance and energy use of lock-free and lock-based implementations of FIFO queues, double-ended queues and sorted singly linked lists. The results from the lock-free and lock-based implementations are compared and also analyzed using captured hardware performance counters, *e.g.* instruction count, user/system time, L1 cache miss ratio and branch misprediction rate. Gautham *et al.* [9] compared the performance and energy use of locks and software transactional memory in benchmarks from the STAMP benchmark suite.

A variety of models have been proposed to estimate power dissipation, based on different approaches. PMC (Performance Monitoring Counters) based power models, build upon event selection and statistical correlation, draw considerable amount of attention. Using this approach, Contreras *et al.* [10] estimated CPU and memory power. Wang *et al.* [11] provided a two level power model for multiprocessors, which uses frequency and IPC (Instructions Per Cycle) as the only PMC event. Isci *et al.* [12] described a technique to estimate per-component power dissipation for CPU using PMCs and used this to determine phases of a program. Tiwari *et al.* [13] created an instruction level power model. They determined a base cost for each instruction type with micro-benchmarks

Procedure Enqueuer

```
1 while ! done do
2 |  el ← Parallel_Work(pw_e);
3 |  Enqueue(el);
```

Procedure Dequeuer

```
1 while ! done do
2 |  el ← Dequeue();
3 |  Parallel_Work(pw_d);
```

Figure 4.1: Thread procedures

and tried to clarify the inter-instruction impacts to estimate power dissipation of compositions. Ge and Cameron [14] provided a power-aware speedup model. They decompose the program into phases according to the degree of available parallelism and on/off-chip access ratios that is used to capture the impact of frequency scaling and process count. Choi *et al.* [15] introduced a roofline model which is parameterized with the maximum throughputs, operation energy and power cap values. They bound the throughput with the power cap, since energy consumption per unit of time depends on throughput, and extract the parameters' values using regression.

As seen above there exist some empirical studies on energy/power consumption of lock-free data structures and a huge variety of power models but we are not aware of any energy model targeting lock-free data-structures. In this study, we aim to begin filling this gap by providing a detailed analysis of power and performance of lock-free queues.

4.3 Framework

4.3.1 Synthetic Benchmark

4.3.1.1 Skeleton

We run the synthetic benchmark composed of the two functions described in Figure 4.1, starting with an empty queue. Half of the threads are assigned to be enqueuers while the remaining ones are dequeuers. We disable logical cores (hyper-threading) and map different threads into different cores, also the number of threads never exceeds the number of cores. In addition, the mapping is

done in the following way: when adding an enqueuer/dequeuer pair, they are both mapped on the most filled but non-full socket.

The parallel sections (Parallel_Work) shall be seen as a processing activity, pre-processing for the enqueuers before they enqueue an item, and post-processing on an item from the queue for the dequeuers. We assume that memory accesses in the parallel sections are negligible, and represent the parallel sections as sequences of bunches of *pause* instructions in the benchmark; we note pw_e (resp. pw_d) the number of bunches of 90 *pause*s (which corresponds to 1000 cycles) that compose the parallel work in the enqueuer (resp. dequeuer).

From a high-level perspective, Enqueue and Dequeue operations follow a retry loop pattern: a thread reads an access point to the data structure, works locally with this view of the data structure, possibly performs memory management actions and prepares the new desired value as an access point of the data structure. Finally, it atomically tries to perform the change through a call to the *Compare-and-Swap* primitive. If it succeeds, *i.e.* if the access point has not been changed by another thread between the first read and the *Compare-and-Swap*, then it goes to the next parallel section, otherwise it repeats the process.

4.3.1.2 Queue Implementations

We study some of the most well-known and studied lock-free and linearizable queues in the literature, as implemented in NOBLE [16]. The legend depicted in Figure 4.2 will be used throughout the paper. The aim of this work is still to predict the behavior of any lock-free queue algorithm and not only the ones mentioned above. These algorithms are used to validate the model that we present in the following sections.

4.3.2 General Power Model

The power is split into three elements: the *static* part is the cost of turning the machine on, the *activation* part incorporates a fixed cost for each socket and each core in use, and the *dynamic* part is a supplementary cost that depends on the running application.

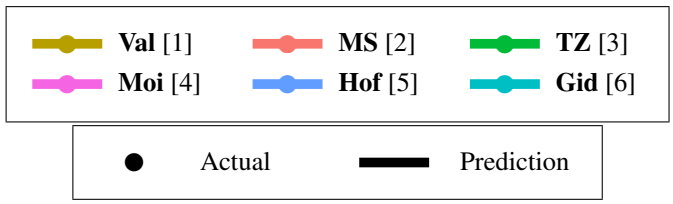


Figure 4.2: Key legend of the graphs

In accordance with the RAPL energy counters [17–19], we further decompose each part per-component, for memory, CPU, and *uncore* (denoted by a superscript M, C and U, respectively):

$$P = \sum_{X \in \{M,C,U\}} \left(P^{(stat,X)} + P^{(active,X)} + P^{(dyn,X)} \right).$$

We assume that we already know the platform characteristics, *i.e.* all static and active powers (they can be obtained as explained for instance in the companion research report [20]), and we try to find the application-specific dynamic powers. In order to keep the formulas readable, in the following, we denote by $P^{(X)}$ the dynamic power $P^{(dyn,X)}$.

4.3.3 Notations and Setting

We denote by n the number of running threads that call the same operation, and by f the clock frequency of the cores (we only consider the case where all cores share the same clock frequency).

We recall that pw_e (resp. pw_d) is the amount of work in the parallel section of an enqueuer (resp. dequeuer), as the number of bunches of 90 *pause*s. For a given queue implementation, we denote by cw_e (resp. cw_d) the amount of work in one try of the retry loop of the Enqueue (resp. Dequeue) operation. Associated with these amounts of work, we define, for $o \in \{d, e\}$, the average execution time of the parallel section (resp. the retry loop and a single try of the retry loop) related to operation o as $\mathrm{t}(PS_o)$ (resp. $\mathrm{t}(RL_o)$ and $\mathrm{t}(SL_o)$).

In the same way, for $o \in \{d, e\}$, we denote by $P_o^{(X)}$ (resp. $P_{o,PS}^{(X)}$ and $P_{o,RL}^{(X)}$) the dynamic power dissipated by component X in (resp. the parallel section related to and the retry loop related to) operation o.

Finally, for $o \in \{d, e\}$, we denote by r_o the ratio of the time that a thread spends in the retry loop, while it is associated with operation o.

In Sections 4.4 and 4.5, in order to keep expressions as simple as possible, we define one unit of time as λ sec, where λ is the execution time of $90 \times f$ *pause*s (as the *pause* instructions are perfectly scalable with clock frequency, λ is constant). Throughput is expressed in number of operations per unit of time, *i.e.* per λ secs. Finally, we derive the power in Watts.

All experiments and their underlying predictions are done on a platform composed of a dual-socket Intel® Xeon® processor, with eight cores per socket. The sizes of L3, L2 and L1 caches are 25 MB, 256 kB and 32 kB, respectively.

We run the implementations at the two extreme frequencies $1.2\,\mathrm{GHz}$ and $3.4\,\mathrm{GHz}$, for all possible even total numbers of threads, from 2 to 16, *i.e.* for $n \in \{1, \ldots, 8\}$.

4.4 Throughput Estimation

4.4.1 Throughput Decomposition Principles

We recall that the throughput of the queue is defined as:

$$\mathcal{T} = \min\left(\mathcal{T}_e, \mathcal{T}_d\right),$$

where $\mathcal{T}_e$ and $\mathcal{T}_d$ are the enqueuers' and dequeuers' throughput, respectively.

As we are in steady state, one operation o is performed every $\mathrm{t}\left(PS_o\right) + \mathrm{t}\left(RL_o\right)$ unit of time by each thread, and n threads attempt to concurrently execute o, hence the general expression of the throughput $\mathcal{T}_o$:

$$\mathcal{T}_o = \frac{n}{\mathrm{t}\left(PS_o\right) + \mathrm{t}\left(RL_o\right)}.$$

We have seen that the parallel sections of the benchmark are full of *pause*s, thus the time $\mathrm{t}\left(PS_o\right)$ spent in a given parallel section is straightforwardly given

by $\mathrm{t}\left(PS_o\right) = pw_o/f$. The execution time of dequeue and enqueue operations is more problematic, for two main reasons. *Primo*, because of the lock-free nature of the implementations. As the number of retries is unknown, the time spent in the function call is not trivially computable. *Secundo*, when the activity on the queue is high, the threads compete for accessing a shared data, and they stall before actually being able to access the data. We name this as the *expansion*, as it leads to an increase in the execution time of a single try of the retry loop.

The contention on the queue is twofold. At any time, and even if it could be negligible, threads that perform the same operation disturb each other, since they try to access the same shared data. In addition, when the queue is mostly empty, enqueuers and dequeuers try to access the same data, then interference occurs; enqueuers make dequeuers stall and *vice versa*. We call the former case *intra-contention*, and the latter one *inter-contention*.

As expected, we have noticed a marked difference between the execution time of a dequeue operation returning NULL and one that returns a queue item, *i.e.* whether the queue was empty or contained at least one item. That is why we decompose $\mathcal{T}_d$ into throughput of dequeue on empty queue $\mathcal{T}_d^{(+)}$ (that returns a NULL item), and dequeue on non-empty queue $\mathcal{T}_d^{(-)}$ (that does not return NULL).

Further, the impact of inter-contention on dequeue operations is negligible compared to the impact of the queue being empty; therefore we ignore inter-contention for dequeues.

In contrast, the queue being empty does not notably change the execution time of the enqueue operation, while dequeue operations can impact the behavior of concurrent enqueue operations greatly when the queue is close to empty. Hence, we split $\mathcal{T}_e$ into the enqueue throughput $\mathcal{T}_e^{(+)}$ when the queue is not inter-contended, and the enqueue throughput $\mathcal{T}_e^{(-)}$ when the queue experiences the maximum possible inter-contention.

These basic throughputs fulfill the two following inequalities: $\mathcal{T}_d^{(+)} \geq \mathcal{T}_d^{(-)}$ and $\mathcal{T}_e^{(+)} \geq \mathcal{T}_e^{(-)}$.

Thanks to this separation into the four basic throughput cases $\mathcal{T}_d^{(+)}$, $\mathcal{T}_d^{(-)}$, $\mathcal{T}_e^{(+)}$ and $\mathcal{T}_e^{(-)}$, we earn a better understanding of the factors that influence the

general throughput, and we deinterlace their dependencies, which dramatically decreases the number of points in the parallel section sizes set where we need to take measurements for our modeling. More precisely, by construction, $\mathcal{T}_d^{(+)}$ and $\mathcal{T}_d^{(-)}$ do not indeed depend on pw_e, while $\mathcal{T}_e^{(+)}$ and $\mathcal{T}_e^{(-)}$ do not depend on pw_d. Nonetheless $\mathcal{T}_d$ (resp. $\mathcal{T}_e$) is defined as a barycenter between $\mathcal{T}_d^{(+)}$ and $\mathcal{T}_d^{(-)}$ (resp. $\mathcal{T}_e^{(-)}$ and $\mathcal{T}_e^{(+)}$), whose weights depend on both pw_d and pw_e.

In Section 4.4.2, we describe the basic throughputs, we combine them in Section 4.4.3, then we explain how to instantiate the parameters of the model in Section 4.4.4, and finally exhibit results in Section 4.4.5.

4.4.2 Basic Throughputs

We aim in this section at estimating the throughput $\mathcal{T}_o^{(b)}$ of one of the basic operations described in the previous subsection, where $o \in \{e, d\}$ and $b \in \{+, -\}$. We assume that $\mathcal{T}_o^{(b)}$ depends only on pw_o, in addition to the tacit dependencies on the clock frequency, number of threads and queue implementation. We denote by $cw_o^{(b)}$ the amount of work in a single try of the retry loop related to operation o in case b when the queue is not intra-contended.

4.4.2.1 Low Intra-Contention

We study in this section the low intra-contention case, *i.e.* when (i) the threads do not suffer from expansion due to threads that perform the same operation, and (ii) a success is obtained with a single try of the retry loop. As it appears in Figure 4.3, we have a cyclic execution, and the length of the shortest cycle is $\mathrm{t}\left(PS_o\right) + \mathrm{t}\left(SL_o^{(b)}\right)$. Within each cycle, every thread performs exactly one successful operation, thus the throughput is easy to compute:

$$\mathcal{T}_o^{(b)} = \frac{n}{\mathrm{t}\left(PS_o\right) + \mathrm{t}\left(SL_o^{(b)}\right)} = \frac{nf}{pw_o + cw_o^{(b)}}. \tag{4.1}$$

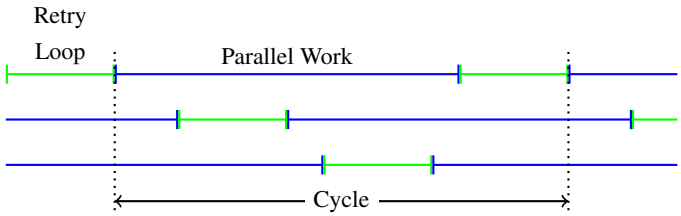


Figure 4.3: Cyclic execution under low intra-contention

4.4.2.2 High Intra-Contention

As explained in Section 4.4.1, in this case, the direct evaluation of the execution time of a retry loop is more complex, but we have experimentally observed that the throughput is approximately linear with the expected number of threads that are in the retry loop at a given time. In addition, this expected number is almost linear to the amount of work in the parallel section. As a result, a good approximation of the throughput, in high intra-contention cases, is a function that is linear with the amount of work in the pw_o.

4.4.2.3 Frontier

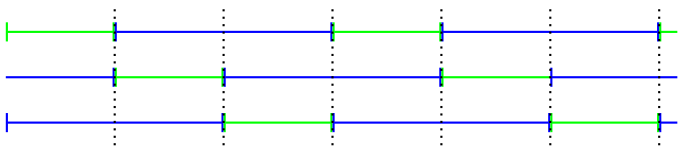


Figure 4.4: Intra-contention frontier

We now have to estimate whether the queue is highly intra-contended.

There exists a simple lower bound of the amount of work in the parallel section, such that there exists an execution where the threads are never failing in their retry loop. We plot in Figure 4.4 an ideal execution with $n = 3$ threads

and $\mathrm{t}\left(PS_o\right) = (n-1) \times \mathrm{t}\left(SL_o^{(b)}\right)$. In this execution, all threads always succeed at their first try in the retry loop. Nevertheless, if we shorten the parallel section, then there is not enough parallel potential any more, and the threads will start to fail: the queue leaves the low intra-contention state.

In practice, this lower bound ($\mathrm{t}\left(PS_o\right) = (n-1) \times \mathrm{t}\left(SL_o^{(b)}\right)$) is actually a good approximation for the critical point where the queue switches its state.

4.4.3 Combining Basic Throughputs

We are given parallel sections sizes, and show how to link the throughput of the four basic operations, with the dequeuers' and enqueuers' throughput. There are two possible states for the queue: either it is mostly empty (*i.e.* some NULL items are dequeued), or it gets larger and larger.

In the first case, some of the dequeues will occur on an empty queue. In 1 unit of time, $\mathcal{T}_e$ items are enqueued. These items are dequeued in $\mathcal{T}_e/\mathcal{T}_d^{(-)}$ units of time (the queue is non-empty while they are dequeued), which leads to a slack of $1 - \mathcal{T}_e/\mathcal{T}_d^{(-)}$, where dequeues of NULL items can take place at a rate $\mathcal{T}_d^{(+)}$, hence the following throughput formula:

$$\mathcal{T}_d = \frac{\mathcal{T}_e}{\mathcal{T}_d^{(-)}} \times \mathcal{T}_d^{(-)} + \left(1 - \frac{\mathcal{T}_e}{\mathcal{T}_d^{(-)}}\right) \times \mathcal{T}_d^{(+)}. \tag{4.2}$$

Concerning the enqueuers, we use the same assumption on inter-contention as used on intra-contention in Section 4.4.2.2, saying that the throughput is linear with the expected number of threads inside the retry loop. Here, the expected number of threads inside the dequeue operation is proportional to the ratio r_d of the time spent by one dequeuer in its dequeue operation. We do not know $\mathrm{t}\left(RL_d\right)$, but we know that in average, to complete a successful operation, a thread needs $\mathrm{t}\left(PS_d\right) + \mathrm{t}\left(RL_d\right)$ units of time, and among this time it will spend $\mathrm{t}\left(PS_d\right)$ in the parallel section. Therefore

$$r_d = 1 - \mathrm{t}\left(PS_d\right) / \left(\mathrm{t}\left(PS_d\right) + \mathrm{t}\left(RL_d\right)\right) = 1 - \frac{\mathcal{T}_d \times pw_d}{n \times f}.$$

The minimum inter-contention is reached when this ratio is 0, while the maxi-

mum is obtained when it is 1, thus:

$$\mathcal{T}_e = \frac{\mathcal{T}_d \times pw_d}{n \times f} \times \mathcal{T}_e^{(+)} + \left(1 - \frac{\mathcal{T}_d \times pw_d}{n \times f}\right) \times \mathcal{T}_e^{(-)}. \tag{4.3}$$

In the second case, enqueuers and dequeuers do not access to the same part of the queue, thus inter-contention does not take place, then $\mathcal{T}_e = \mathcal{T}_e^{(+)}$, and all dequeues return a non-NULL item, hence $\mathcal{T}_d = \mathcal{T}_d^{(-)}$.

The discrimination of these two cases is trivial when enqueuers' and dequeuers' throughput are given: the queue is in the first state (mostly empty) if and only if $\mathcal{T}_e \leq \mathcal{T}_d$.

Reversely, if we know the four basic throughputs, and aim at reconstituting the dequeuers' and enqueuers' throughput, several solutions could be consistent.

Theorem 1. *Given* $\left(\mathcal{T}_e^{(+)}, \mathcal{T}_e^{(-)}, \mathcal{T}_d^{(+)}, \mathcal{T}_d^{(-)}\right)$*, there exists a solution* $(\mathcal{T}_d, \mathcal{T}_e)$ *with a growing queue if and only if* $\mathcal{T}_e^{(+)} > \mathcal{T}_d^{(-)}$*. In addition, this solution is unique and is such that* $\mathcal{T}_e = \mathcal{T}_e^{(+)}$ *and* $\mathcal{T}_d = \mathcal{T}_d^{(-)}$*.*

Proof. ($\Rightarrow$) If the queue is growing, then $\mathcal{T}_e > \mathcal{T}_d$. Moreover, dequeues never occur on an empty queue, hence $\mathcal{T}_d = \mathcal{T}_d^{(-)}$, and there is no inter-contention, thus $\mathcal{T}_e = \mathcal{T}_e^{(+)}$.
($\Leftarrow$) Let us assume now that $\mathcal{T}_e^{(+)} > \mathcal{T}_d^{(-)}$. $\mathcal{T}_e = \mathcal{T}_e^{(+)}$ and $\mathcal{T}_d = \mathcal{T}_d^{(-)}$ is a valid solution, such that the queue is growing, since then $\mathcal{T}_e > \mathcal{T}_d$.

By construction, $\mathcal{T}_e \leq \mathcal{T}_e^{(+)}$; if we had another solution such that the queue grows and $\mathcal{T}_e < \mathcal{T}_e^{(+)}$, it would mean that enqueues are inter-contended, which is possible only when the queue is mostly empty. This is absurd, hence the uniqueness. □

Theorem 2. *Given* $\left(\mathcal{T}_e^{(+)}, \mathcal{T}_e^{(-)}, \mathcal{T}_d^{(+)}, \mathcal{T}_d^{(-)}\right)$*, there exists a solution* $(\mathcal{T}_d, \mathcal{T}_e)$ *with a mostly empty queue if and only if*

$$\frac{\mathcal{T}_e^{(-)}}{\mathcal{T}_d^{(-)}} \leq 1 - \frac{pw_d}{n \times f}\left(\mathcal{T}_e^{(+)} - \mathcal{T}_e^{(-)}\right). \tag{4.4}$$

In addition, this solution is unique and is given by Equations 4.3 and 4.2.

Proof. ($\Rightarrow$) Let a solution with a mostly empty queue. By construction, the throughputs follow Equations 4.3 and 4.2. As $\mathcal{T}_e$ is an increasing function according to $\mathcal{T}_d$ (because $\mathcal{T}_e^{(+)} \geq \mathcal{T}_e^{(-)}$), we derive

$$\mathcal{T}_e \geq \frac{\mathcal{T}_d^{(-)} \times pw_d}{n \times f} \times \mathcal{T}_e^{(+)} + \left(1 - \frac{\mathcal{T}_d^{(-)} \times pw_d}{n \times f}\right) \times \mathcal{T}_e^{(-)}.$$

The queue is mostly empty, thus the dequeues of non-NULL items have to be faster than the enqueues, which translates into $\mathcal{T}_d^{(-)} \geq \mathcal{T}_e$. The two inequalities combined show the implication.
($\Leftarrow$) Let us assume now that Inequality 4.4 is fulfilled. Equation 4.2 can be rewritten into

$$\mathcal{T}_e = \frac{\mathcal{T}_d - \mathcal{T}_d^{(+)}}{1 - \frac{\mathcal{T}_d^{(+)}}{\mathcal{T}_d^{(-)}}}.$$

Let us consider now $\mathcal{T}_e{}'$ and $\mathcal{T}_e{}''$ two functions of $\mathcal{T}_d{}'$ that fulfill the following system of equations:

$$\begin{cases} \mathcal{T}_e{}'\left(\mathcal{T}_d{}'\right) = \frac{\mathcal{T}_d{}' - \mathcal{T}_d^{(+)}}{1 - \frac{\mathcal{T}_d^{(+)}}{\mathcal{T}_d^{(-)}}} \\ \mathcal{T}_e{}''\left(\mathcal{T}_d{}'\right) = \frac{\mathcal{T}_d{}' \times pw_d}{n \times f} \times \mathcal{T}_e^{(+)} + \left(1 - \frac{\mathcal{T}_d{}' \times pw_d}{n \times f}\right) \times \mathcal{T}_e^{(-)}. \end{cases}$$

We have $\mathcal{T}_e{}'\left(\mathcal{T}_d^{(+)}\right) = 0$ and $\mathcal{T}_e{}'\left(\mathcal{T}_d^{(-)}\right) = \mathcal{T}_d^{(-)}$. According to Inequality 4.4, we know also that $\mathcal{T}_e{}''\left(\mathcal{T}_d^{(-)}\right) \leq \mathcal{T}_d^{(-)}$. In addition, $\mathcal{T}_e{}''$ is a linearly increasing function of $\mathcal{T}_d{}'$ and $\mathcal{T}_e{}'$ a linearly decreasing function of $\mathcal{T}_d{}'$. This shows that there exists a unique $\mathcal{T}_d$ such that $\mathcal{T}_e{}'\left(\mathcal{T}_d\right) = \mathcal{T}_e{}''\left(\mathcal{T}_d\right)$, and if we define $\mathcal{T}_e$ as $\mathcal{T}_e = \mathcal{T}_e{}'\left(\mathcal{T}_d\right) = \mathcal{T}_e{}''\left(\mathcal{T}_d\right)$, the pair $(\mathcal{T}_d, \mathcal{T}_e)$ is such that

$$\begin{cases} \mathcal{T}_d^{(-)} \leq \mathcal{T}_d \leq \mathcal{T}_d^{(+)} \\ \mathcal{T}_e^{(-)} \leq \mathcal{T}_e \leq \mathcal{T}_e^{(+)} \\ \mathcal{T}_e \leq \mathcal{T}_d \end{cases}.$$

This implies that it is a solution with an empty queue, and we have shown that this solution is unique. □

Corollary 1. *Given* $\left(\mathcal{T}_e^{(+)}, \mathcal{T}_e^{(-)}, \mathcal{T}_d^{(+)}, \mathcal{T}_d^{(-)}\right)$*, there exists at least one solution* $(\mathcal{T}_d, \mathcal{T}_e)$.

Proof. We show that if the inequality of Theorem 1 is not fulfilled, *i.e.* if $\mathcal{T}_e^{(+)} \leq \mathcal{T}_d^{(-)}$, then the inequality of Theorem 2 is true. We have indeed

$$
\begin{aligned}
&\mathcal{T}_d^{(-)} \times \left(1 - \frac{pw_d}{n \times f}\left(\mathcal{T}_e^{(+)} - \mathcal{T}_e^{(-)}\right)\right) - \mathcal{T}_e^{(-)} \\
=&\mathcal{T}_d^{(-)} \times \left(1 - \frac{pw_d \times \mathcal{T}_e^{(+)}}{n \times f}\right) - \mathcal{T}_e^{(-)} \times \left(1 - \frac{pw_d \times \mathcal{T}_d^{(-)}}{n \times f}\right) \\
\geq&\mathcal{T}_d^{(-)} \times \left(1 - \frac{pw_d \times \mathcal{T}_e^{(+)}}{n \times f}\right) - \mathcal{T}_e^{(+)} \times \left(1 - \frac{pw_d \times \mathcal{T}_d^{(-)}}{n \times f}\right) \\
\geq&\mathcal{T}_d^{(-)} - \mathcal{T}_e^{(+)} \\
&\mathcal{T}_d^{(-)} \times \left(1 - \frac{pw_d}{n \times f}\left(\mathcal{T}_e^{(+)} - \mathcal{T}_e^{(-)}\right)\right) - \mathcal{T}_e^{(-)} \geq 0,
\end{aligned}
$$

which proves the Corollary. □

One can notice that if $\mathcal{T}_e^{(+)} > \mathcal{T}_d^{(-)}$ and Inequality 4.4 are fulfilled and the queue could be either mostly empty or growing. In this case, we choose, for each operation, the mean of the two solutions, in order to minimize the discontinuities.

4.4.4 Instantiating the Throughput Model

We recall that, for all o and b, $\mathcal{T}_o^{(b)}$ depends only on pw_o, while $\mathcal{T}_e$ and $\mathcal{T}_d$ depend on both pw_d and pw_e. We denote now by $\mathcal{T}_d(pw_d, pw_e)$ (respectively $\mathcal{T}_e(pw_d, pw_e)$) the dequeuers' (respectively enqueuers') throughput as the amount of work in the parallel section of the dequeuers is pw_d and enqueuers' one is pw_e. The estimate of a value is denoted by a hat on top, while the measured value does not wear the hat.

Let $p_s = 1$, $p_m = 20$ and $p_b = 1000$ be three distinctive amounts of work, that corresponds to different states of the execution. If $pw_o = p_b$, we can neglect the impact of operation o on the queue, $pw_o = p_m$ is a low intra-contention case since the non-expanded critical sections are experimentally less

than 2 units of time, and $pw_o = p_s$ corresponds to a highly inter- or intra-contention case. We note the we cannot use a 0 size as amount of work since it leads to undesirable results due to the back-to-back effect (a thread does not allow other threads to access the queue for several consecutive iterations).

4.4.4.1 Low Intra-Contention

The basic throughputs that are not intra-contended can be spawned from $cw_o^{(b)}$, which we try to estimate here. We pick four points where the basic throughputs are easy to approximate. We have $\mathcal{T}_d(p_m, p_s) < \mathcal{T}_e(p_m, p_s)$, as the order of magnitude of the amounts of work in the retry loops is less than a few units. For the same reason, at this point, we are in low intra-contention from the dequeuers' point of view. Altogether,

$$\mathcal{T}_d(p_m, p_s) = \mathcal{T}_d^{(-)}(p_m) = \frac{n \times f}{p_m + cw_d^{(-)}}, \text{ hence}$$

$$\widehat{cw_d^{(-)}} = \frac{n \times f}{\mathcal{T}_d(p_m, p_s)} - p_m.$$

Then, according to Equation 4.2, we have

$$\frac{nf}{p_m + \widehat{cw_d^{(+)}}} = \mathcal{T}_d^{(+)}(p_m)$$

$$\frac{nf}{p_m + \widehat{cw_d^{(+)}}} = \frac{\mathcal{T}_d(p_m, p_b) - \mathcal{T}_e(p_m, p_b)}{1 - \frac{\left(p_m + \widehat{cw_d^{(-)}}\right) \times \mathcal{T}_e(p_m, p_b)}{n \times f}},$$

from which we can extract $\widehat{cw_d^{(+)}}$ since we know already $\widehat{cw_d^{(-)}}$.

In the same way, we can compute $\widehat{cw_e^{(+)}}$ then $\widehat{cw_e^{(-)}}$, by using (p_b, p_m) and (p_s, p_m).

4.4.4.2 High Intra-Contention

We aim here at estimating $\mathcal{T}_o^{(b)}$ on a high intra-contention point. $p_s = 1$ and $p_m = 20$ are such that $\mathcal{T}_d(p_s, p_m) \geq \mathcal{T}_e(p_s, p_m)$. According to Equation 4.2,

we have

$$\mathcal{T}_d(p_s, p_m) = \mathcal{T}_e(p_s, p_m) + \left(1 - \frac{\mathcal{T}_e(p_s, p_m)}{\widehat{\mathcal{T}_d^{(-)}}(p_s)}\right) \times \widehat{\mathcal{T}_d^{(+)}}(p_s).$$

In addition, if $\mathcal{T}_d(p_s, p_s) \geq \mathcal{T}_e(p_s, p_s)$, then

$$\mathcal{T}_d(p_s, p_s) = \mathcal{T}_e(p_s, p_s) + \left(1 - \frac{\mathcal{T}_e(p_s, p_s)}{\widehat{\mathcal{T}_d^{(-)}}(p_s)}\right) \times \widehat{\mathcal{T}_d^{(+)}}(p_s),$$

otherwise, $\mathcal{T}_d(p_s, p_s) = \widehat{\mathcal{T}_d^{(-)}}(p_s)$. In both cases, we can find the two unknowns $\widehat{\mathcal{T}_d^{(-)}}(p_s)$ and $\widehat{\mathcal{T}_d^{(+)}}(p_s)$ thanks to the two equations.

This last point is also used in the same way for enqueuers: if $\mathcal{T}_d(p_s, p_s) \geq \mathcal{T}_e(p_s, p_s)$, then

$$\mathcal{T}_e(p_s, p_s) = \frac{\mathcal{T}_d(p_s, p_s) \times p_s}{n \times f} \times \widehat{\mathcal{T}_e^{(+)}}(p_s) + \left(1 - \frac{\mathcal{T}_d(p_s, p_s) \times p_s}{n \times f}\right) \times \widehat{\mathcal{T}_e^{(-)}}(p_s),$$

otherwise, $\mathcal{T}_e(p_s, p_s) = \widehat{\mathcal{T}_e^{(+)}}(p_s)$.

Like previously, we have $\mathcal{T}_d(p_m, p_s) < \mathcal{T}_e(p_m, p_s)$, hence $\widehat{\mathcal{T}_e^{(+)}}(p_s) = \mathcal{T}_e(p_m, p_s)$. This implies that in any cases we can compute $\widehat{\mathcal{T}_e^{(+)}}(p_s)$, but we do not have access to $\widehat{\mathcal{T}_e^{(-)}}(p_s)$ if $\mathcal{T}_d(p_s, p_s) < \mathcal{T}_e(p_s, p_s)$. In this case, the bottleneck of the queue is likely to be the dequeuers, hence we set the value $\widehat{\mathcal{T}_e^{(-)}}(p_s) = \widehat{\mathcal{T}_e^{(+)}}(p_s)$ by default.

All $\widehat{\mathcal{T}_o^{(b)}}$ are then obtained by joining $\widehat{\mathcal{T}_o^{(b)}}(p_s)$ to the leftmost point of the low intra-contention part:

$$\widehat{\mathcal{T}_o^{(b)}}(pw_o) = \begin{cases} \frac{\frac{f}{\widehat{cw_o^{(b)}}} - \widehat{\mathcal{T}_o^{(b)}}(p_s)}{(n-1)\widehat{cw_o^{(b)}} - p_s} \times (pw_o - p_s) + \widehat{\mathcal{T}_o^{(b)}}(p_s) \\ \qquad \text{if } pw_o \leq (n-1)\widehat{cw_o^{(b)}} \\ \\ \frac{n \times f}{pw_o + \widehat{cw_o^{(b)}}} \quad \text{otherwise.} \end{cases}$$

Finally, dequeuers' and enqueuers' throughput are reconstituted as explained in Section 4.4.3: if Equation 4.4 is fullfilled, then they are computed through Equations 4.2 and 4.3 that can be rewritten as:

$$\begin{cases} \widehat{\mathcal{T}_d}(pw_d, pw_e) = \dfrac{\widehat{\mathcal{T}_d^{(+)}}(pw_d) + \widehat{\mathcal{T}_e^{(-)}}(pw_e)\left(1 - \frac{\widehat{\mathcal{T}_d^{(+)}}(pw_d)}{\widehat{\mathcal{T}_d^{(-)}}(pw_d)}\right)}{1 - \frac{pw_d}{nf}\left(\widehat{\mathcal{T}_e^{(+)}}(pw_e) - \widehat{\mathcal{T}_e^{(-)}}(pw_e)\right)\left(1 - \frac{\widehat{\mathcal{T}_d^{(+)}}(pw_d)}{\widehat{\mathcal{T}_d^{(-)}}(pw_d)}\right)} \\ \\ \widehat{\mathcal{T}_e}(pw_d, pw_e) = \frac{\widehat{\mathcal{T}_d}(pw_d, pw_e) \times pw_d}{n \times f} \times \widehat{\mathcal{T}_e^{(+)}}(pw_e) \\ \qquad\qquad + \left(1 - \frac{\widehat{\mathcal{T}_d}(pw_d, pw_e) \times pw_d}{n \times f}\right) \times \widehat{\mathcal{T}_e^{(-)}}(pw_e). \end{cases}$$

Otherwise, $\widehat{\mathcal{T}_d}(pw_d, pw_e) = \widehat{\mathcal{T}_d^{(-)}}(pw_d)$ and $\widehat{\mathcal{T}_e}(pw_d, pw_e) = \widehat{\mathcal{T}_e^{(+)}}(pw_e)$.

4.4.5 Results

The throughput predictions are plotted in Figures 4.5 and 4.6 for the enqueuers, and in Figure 4.7 for the dequeuers (the legend is in Figure 4.2). Points are measurements, while lines are predictions. We will follow this rule for all comparisons between prediction and measurement. In the actual execution, the queue goes through a transient state when the amount of work in the parallel section is near the critical point, but the prediction is not so far from the actual measurements, as illustrated in Figures 4.5 and 4.6. Under intra-contention, some of the curves get flat, since only one thread can be succeeding at the same time,

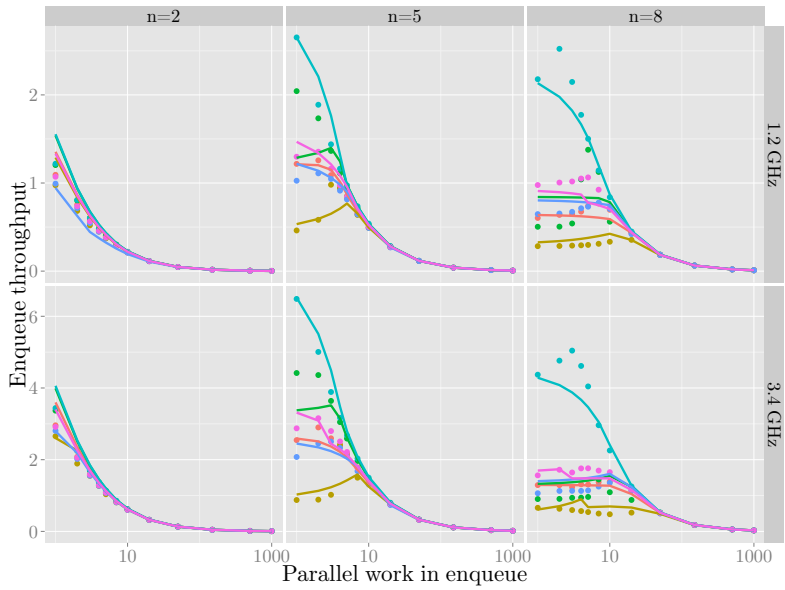


Figure 4.5: Enqueue throughput with $pw_d = 7$

according to the definition of the retry loop. Some curves even decrease because the successful one is stalled by other failing ones due to serialization of the atomic primitives, namely expansion. The slope presumably indicates the density of atomic primitives in retry loops which depends on the algorithm.

The comparison of Figures 4.5 and 4.6 illustrates the impact of inter-contention. A decrease of the highest point of $\mathcal{T}_e$, due to an increase of cw_e, can be observed for the more inter-contended case. When cw_e increases, some critical points shift slightly towards the right as the intra-contention starts with a larger pw_e. In Figure 4.7, decomposition of $\mathcal{T}_d$ is apparent. When enqueue rate is low, *i.e.* when pw_e is high, $\mathcal{T}_d$ is ruled by $\mathcal{T}_d^{(+)}$ due to majority of NULL dequeues, and it tends towards $\mathcal{T}_d^{(-)}$ when the enqueue rate increases.

Graphs on a wider set of parameters are available in the companion research report [20], in the form of animated figures.

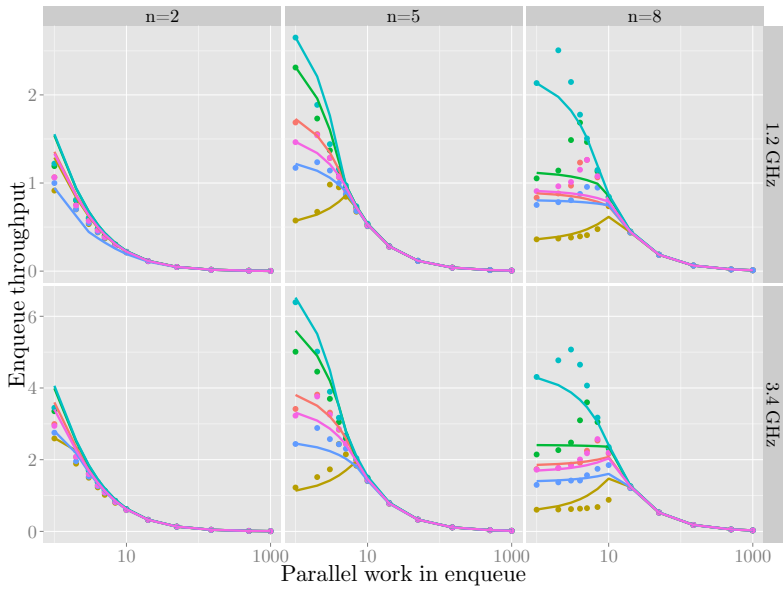


Figure 4.6: Enqueue throughput with $pw_d = 50$

4.5 Power Estimation

We recall that we are interested only in the dynamic powers as we assume that static and activation powers are known.

4.5.1 CPU Power

Firstly, as we map each thread on a dedicated core, there is no interference between the CPU power of different cores, so we can compute the dynamic power as

$$P^{(C)} = n \times P_e^{(C)} + n \times P_d^{(C)}. \tag{4.5}$$

Secondly, we assume that we can segment time and consider that, given a thread performing operation o, the power dissipated in the retry loop and the

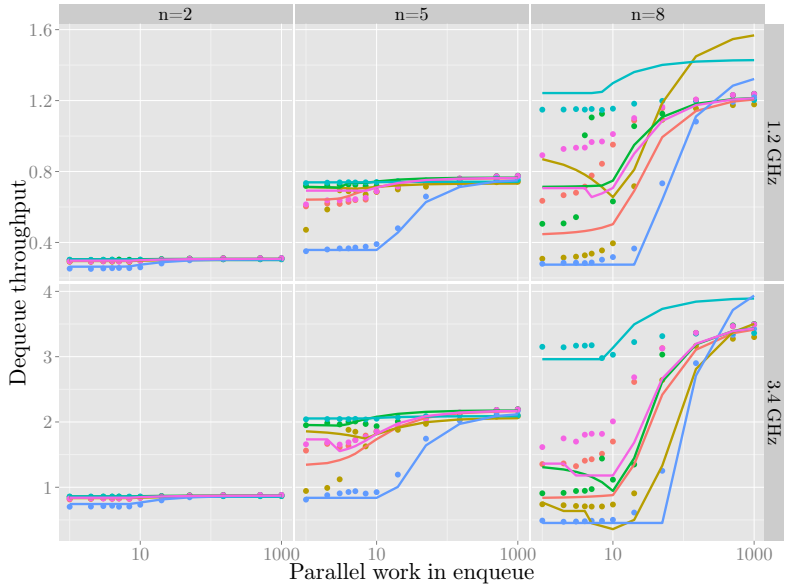


Figure 4.7: Dequeue throughput with $pw_d = 7$

power dissipated in the parallel section are independent. There only remains to weight the previous powers by the time spent in each of these regions:

$$P_o^{(C)} = r_o \times P_{o,RL}^{(C)} + (1 - r_o) \times P_{o,PS}^{(C)}. \tag{4.6}$$

As shown in Section 4.4.3, the ratio can be obtained through

$$r_o = 1 - \frac{\mathcal{T}_o \times pw_o}{n \times f}. \tag{4.7}$$

Altogether, we obtain the final formula for dynamic CPU power

$$P^{(C)} = n \left(\sum_{o \in \{e,d\}} P_{o,RL}^{(C)} + \frac{\mathcal{T}_o \times pw_o \times \left(P_{o,PS}^{(C)} - P_{o,RL}^{(C)} \right)}{n \times f} \right) \tag{4.8}$$

4.5.2 Memory and Uncore Power

We have noticed in [20] that the dynamic memory power is proportional to the intensity (number of units of memory accessed per unit of time) of main memory accesses and remote accesses, when the threads read separate places of the memory.

Here, the data structure does not directly involve the main memory since we keep its size reasonably bounded (if the queue reaches the maximum size, we suspend the measurements, empty the queue, and resume), hence the power dissipation in memory is only due to remote accesses, which only appears as the threads are spread across sockets (*i.e.* when $n > 4$).

Moreover, as the parallel sections are full of *pause*s, communications can only take place in the retry loop, and there is no dynamic memory power dissipated in the parallel sections. Concerning the retry loops, we make the following assumption: the amount of data accessed per second in a retry loop depends on the implementation, but given an implementation, once a thread is in the retry loop, it will always try to access the same amount of data per second. When the queue is highly intra-contended, if a thread fails then it will retry and will access the data in the same way as in the previous try; and if there is expansion, then the thread will still try to access the data for the whole time it is in the retry loop.

In addition, the dequeuers (and the same line of reasoning holds for the enqueuers) tries here to access the same data. Therefore either memory requests are batched together when sent outside the socket, or the Home Agent keeps track of the previous requests. This implies that the number of threads attempting to access the data does not impact the dynamic memory power greatly when the rate of requests is high.

All things considered, as a thread working on operation o spends a fraction r_o of its time inside its retry loop, we obtain that the dynamic memory power dissipated in the retry loop is proportional to r_o (times the amount of data accessed per unit of time in the retry loop, which is a constant). Hence

$$P^{(M)} = r_e \times \rho_e^{(M)} + r_d \times \rho_d^{(M)}, \tag{4.9}$$

where $\rho_e^{(M)}$ and $\rho_d^{(M)}$ are constants.

The dynamic uncore power is computed exactly in the same way as the dynamic memory power.

4.5.3 Instantiating the Power Model

We use once again $p_s = 1$, $p_m = 20$ and $p_b = 1000$ as three distinctive amounts of work, that allows easy approximations for the power dissipation expressions.

We have seen that if $X \in \{M, U\}$, then $P^{(X)} = r_d \times \rho_d^{(X)} + r_e \times \rho_e^{(X)}$, which can be approximated at $(pw_d, pw_e) = (p_b, p_s)$ by $P^{(X)}(p_b, p_s) = r_e(p_s) \times \rho_e^{(X)}$, since r_d is then nearly 0. It implies that

$$\widehat{\rho_e^{(X)}} = \frac{P^{(X)}(p_b, p_s)}{1 - \frac{\mathcal{T}_e(p_b, p_s) \times p_s}{n \times f}}.$$

We obtain $\widehat{\rho_d^{(X)}}$ similarly at $(pw_d, pw_e) = (p_s, p_b)$.

Concerning the dynamic CPU power, we firstly estimate the power dissipated in the parallel sections. According to the implementation, the CPU power dissipated by the parallel section of enqueuers and dequeuers is the same for both, and this power does not depend on the amount of work. These restrictions are not a loss of generality, since the aim here is to study the queue implementations. It can then be estimated by using (p_b, p_b), where the ratios r_o can be considered as 0, which leads to

$$\widehat{P_{o,PS}^{(C)}} = \frac{P^{(C)}(p_b, p_b)}{2n}.$$

We reuse the point (p_b, p_s), where r_d is very close to 0, to derive that

$$P^{(C)} = n \left(r_e(p_s) \times \widehat{P_{e,RL}^{(C)}} + (1 - r_e(p_s)) \widehat{P_{e,PS}^{(C)}} \right) + n \widehat{P_{d,PS}^{(C)}},$$

which is equivalent to

$$\widehat{P_{e,RL}^{(C)}} = \frac{P^{(C)}(p_b, p_s)}{n \left(1 - \frac{\mathcal{T}_e(p_b, p_s) p_s}{n \times f} \right)} - \left(\frac{2}{1 - \frac{\mathcal{T}_e(p_b, p_s) p_s}{n \times f}} - 1 \right) \widehat{P_{o,PS}^{(C)}}$$

Once again, we obtain $\widehat{P^{(C)}_{d,RL}}$ with the same line of reasoning at $(pw_d, pw_e) = (p_s, p_b)$.

Finally, $\widehat{P^{(M)}}$ and $\widehat{P^{(U)}}$ (resp. $\widehat{P^{(C)}}$) are computed by using Equation 4.9 (resp. Equations 4.5 and 4.6), and the estimates of the ratios that are issued from Section 4.4.

4.5.4 Results

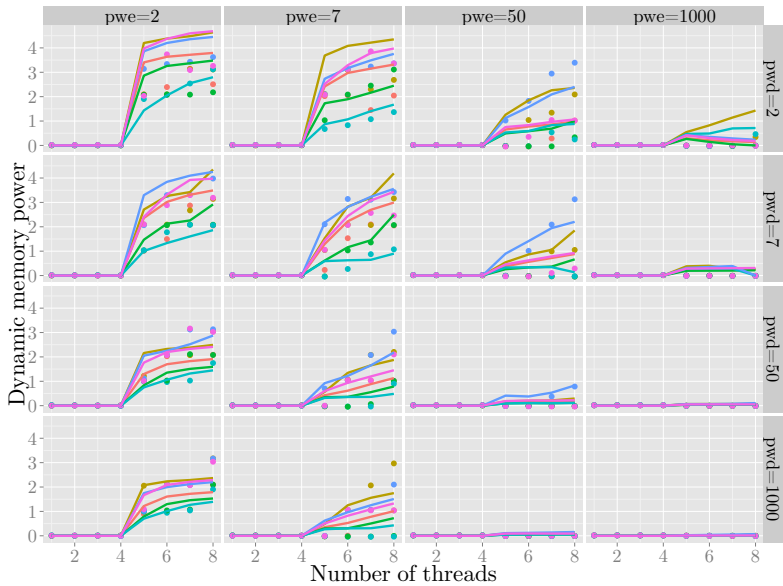


Figure 4.8: Dynamic memory power at $f = 3.4\,\mathrm{GHz}$

As the retry loop, which is particular to each implementation, is mainly composed of memory operations, the main difference between the various implementations in terms of power occurs in the dynamic memory power, which we represent in Figure 4.8 (legend is in Figure 4.2).

Overall, the prediction reacts correctly to the variations of parallel section sizes, and some specifics of the algorithms are caught, *e.g.* **Hof** detached from

the others when $pw_e = 50$ or **Gid** mostly well-predicted both absolutely and relatively as the less power-dissipating implementation.

One can observe once again the asymmetry between enqueue and dequeue operations by comparing the power values at $(pw_d, pw_e) = (2, 1000)$ and $(1000, 2)$; this asymmetry is predicted by the model, with a lower impact though.

Other power comparisons can be found in the companion research report [20], along with the results about the last metric, namely energy per operation.

4.6 Towards Realistic Applications

The performance and energy behavior of an application using a lock-free queue depends on both the application specific code and the implementation of the data structure. For applications where the queue is used in a steady state manner, predictions can be made using the model instantiated with the synthetic benchmark, combined with information about the behavior of the application specific code. What is needed is:

- The size of the parallel work part of the application, both for enqueuers and dequeuers. These may be distributions rather than single values.
- The dynamic power for these parts (as it may differ from that of the parallel work in the synthetic benchmark).

4.6.1 Description of Mandelbrot Set Application

As a case-study we have used an existing application[1] that computes and renders an 8192×8192 pixel image of the Mandelbrot set [22] in parallel using the producer/consumer pattern. The program uses a concurrent queue to communicate between two major phases:

- Phase 1 consists of computing the number (with a maximum of 255) of iterations for a given set of points within a chosen region of the image. The results for each region together with its coordinates are then enqueued.

[1] Previously used for evaluation in [21].

- Phase 2 consists of, for each region dequeued from the queue, computing the RGB values for each contained point and draw these pixels to the resulting image.

Half of the threads perform phase 1 and the rest perform phase 2. The size of each square region is chosen to be one of 16×16, 4×4, or 2×2 pixels which also determines the amount of work to perform per queue operation and, hence, the level of contention. Similarly to the synthetic benchmark, the application uses a dense pinning strategy, pinning producer/consumer pairs to consecutive pairs of cores.

4.6.2 Mandelbrot Prediction

There are two main differences between the Mandelbrot application and the synthetic benchmark: (i) the instructions in the parallel section differ; and (ii) the size of the parallel section for producers varies in Mandelbrot.

Firstly, we need to measure the CPU power dissipation for Mandelbrot; we cannot expect to be able to predict the power dissipation of any application that uses a queue without having any knowledge about the power characteristics of the application. In contrast, memory power dissipation for the computation intensive Mandelbrot parallel section is negligible in comparison to queue operations; hence, the dynamic memory power that we have measured and extrapolated in the synthetic benchmark is unchanged.

Secondly, Mandelbrot provides a variety of producer parallel works. To deal with this, the pixel region is decomposed row-wise in an interleaved manner among producer threads. This decomposition leads to long enough execution intervals in which the parallel sections of the producer threads are similar and constant. This is due to the computationally expensive pixels belonging to the Mandelbrot set being concentrated together in the center of the domain and surrounded by cheaper pixels which diverge quickly. This characteristic is congruent with our model where the data structure is used in a steady state manner. Thus, predictions can be made using the instantiated model over a linear combination of execution intervals.

We measure the latency of the computation intensive producer and consumer parallel works for each frequency and contention level (2×2, 4×4, 16×16). For this process, we make use of CPUID, RDTSC and RDTSCP instructions as specified in [23]. The distribution of parallel works reveals that there are two main groups for producers, that corresponds to regions belonging to the Mandelbrot set or not. Concerning 2×2 contention, due to the wide distribution, we gather the parallel works into bins of width 10 pauses; the number of elements in the i^{th} bin is then denoted by $size^{(i)}$ and its average amount of work by $pw_e^{(i)}$. We scale the width of bins linearly with the area of the region for other contention levels. For the consumers, parallel works are similar for the whole execution.

To make predictions, we assume that all consumer/producer pair $(pw_d, pw_e^{(i)})$ is executed in a steady state during an interval of time. For each frequency, thread, algorithm and contention of interest, we obtain the throughput $\mathcal{T}^{(i)} = \mathcal{T}(pw_d, pw_e^{(i)})$ and the powers $P_i^{(X)} = P^{(X)}(pw_d, pw_e^{(i)})$ for this interval from the corresponding synthetic benchmark input. The only part of the model, instantiated with the synthetic benchmark that needs to be replaced by an application specific entry, is the dynamic CPU power parameter. Then, we combine intervals to obtain total execution time and average power dissipation. This accumulation strategy should be applied with care as the synthetic benchmark is based upon the steady state assumption. An interval which is assumed to take place with a mostly empty queue, could actually not be in this state due to left-over items from the previous interval. Although our model is capable of taking this initial state into consideration and provide metrics accordingly, we assume that each interval is independent. This approximation is reasonable since the consumer parallel work corresponds to the producer bin with one of smallest values, hence a mostly empty queue.

Note that we have implemented a constant back-off equivalent to the consumer parallel work, after dequeuing a NULL item instead of retrying immediately, because of several advantages. It cannot decrease the performance, since either the queue is growing, and then the back-off never takes place, or the queue is mostly empty, and then the producers are the bottleneck of the queue. Con-

versely, it can increase the performance by diminishing the queue contention. Those motivations drove the design of the synthetic benchmark, that we can accordingly reuse here.

For each frequency, thread, algorithm and contention configuration, execution time and power estimates for Mandelbrot application are obtained with the following equations:

$$Time_{total} = \sum_{i=1}^{BinCount} size^{(i)} \times \frac{\lambda}{\mathcal{T}^{(i)}}$$

$$P^{(X)} = \frac{\sum_{i=1}^{BinCount} (size^{(i)} \times \frac{\lambda}{\mathcal{T}^{(i)}}) \times P_i^{(X)}}{Time_{total}}$$

CPU power estimation is straightforward and memory power results are very similar to the synthetic benchmark in Figure 4.8, so we just present and discuss them in [20].

In Figure 4.9, execution time estimates catch the queue algorithm specific trend for high contention cases, which exhibit a more complicated behavior than the low contention cases. Also, they reveal the impact of different queue implementations to overall application performance, which does not appear under low contention. For the highest contention level with region size 2×2, an increasing trend in execution time is observed after 8 threads for many algorithms. The reason is the increasing latency of atomic synchronization primitives originating from two main sources: (i) inter-socket communication, which starts after 8 threads due to our pinning strategy, and (ii) the increasing serialization (expansion) probability for atomic primitives due to increasing number of threads that interfere in the retry loop. The ratio of atomic primitives and the size of queue operations show variations between algorithms which in turn leads to different behaviors. For the 4×4 contention case, the difference between algorithms can still be observed but the parallel sections are large enough to avoid interference in the retry loop. Therefore, execution time decreases with the increasing number of threads. The difference between algorithms is due to

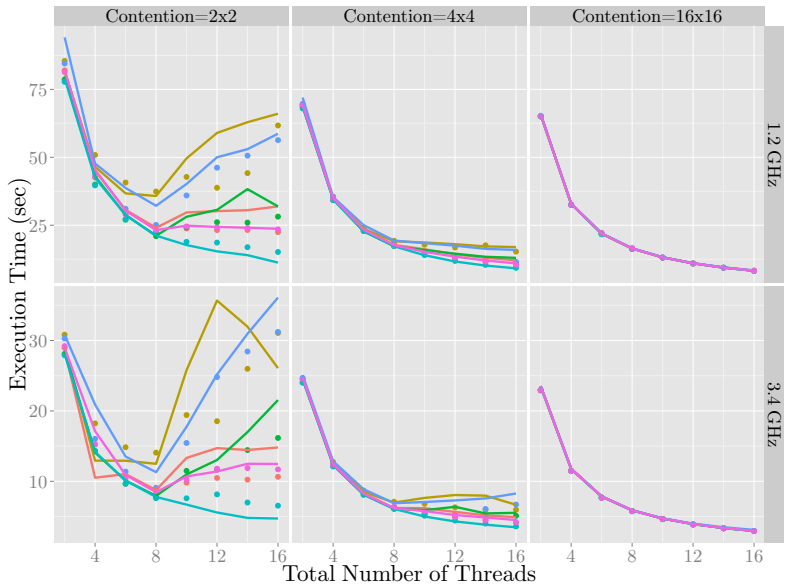


Figure 4.9: Mandelbrot Execution Time

different queue operation sizes which loses its significance gradually with the decreasing contention level, as observed in low contention cases.

4.7 Conclusion

In this paper we have:
(i) proposed models for predicting the throughput and power behavior of lock-free concurrent queues under steady state usage;
(ii) shown how these models can be instantiated for the queue implementations and machine on hand using 10 measurements per frequency and number of threads via a synthetic benchmark; and
(iii) demonstrated that the energy behavior of a parallel application that uses a lock-free queue in a steady state manner can be predicted using these models and only a small amount of queue-implementation-independent empirical

information about the application.

As a future work, it would be of interest to study the strength of the model that has been presented here by testing it on other applications, in particular on more memory-intensive ones.

Furthermore, the model can hopefully be extended to several directions. While staying focused on the queue data structure, lock-based implementations may be included, and behave in a similar way as their lock-free counterparts. To conclude, it would be interesting to generalize the model to other data types.

Bibliography

[1] J. D. Valois, "Implementing Lock-Free Queues," in *Proceedings of International Conference on Parallel and Distributed Systems (ICPADS)*, December 1994, pp. 64–69.

[2] Maged M. Michael and Michael L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the ACM Symposium on Principles of Distributed Computing (PoDC)*. 1996, pp. 267–275, ACM.

[3] Philippas Tsigas and Yi Zhang, "A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2001, pp. 134–143, ACM.

[4] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit, "Using elimination to implement scalable and lock-free fifo queues," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2005, pp. 253–262, ACM.

[5] Moshe Hoffman, Ori Shalev, and Nir Shavit, "The baskets queue," in *Proceedings of the International Conference on Principle of Distributed Systems (OPODIS)*. 2007, pp. 401–414, Springer.

[6] Anders Gidenstam, Håkan Sundell, and Philippas Tsigas, "Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency," in *Proceedings of the International Conference on Principle of Distributed Systems (OPODIS)*. 2010, pp. 302–317, Springer.

[7] Jack J. Dongarra and Peter H. Beckman, "The international exascale software roadmap," *International Journal of High Performance Computing Applications (IJHPCA)*, vol. 25, no. 1, pp. 3–60, 2011.

[8] Nicholas Hunt, Paramjit Singh Sandhu, and Luis Ceze, "Characterizing the performance and energy efficiency of lock-free data structures," in *Workshop on Interaction between Compilers and Computer Architectures (INTERACT)*. 2011, pp. 63–70, IEEE Computer Society.

[9] Ashok Gautham, Kunal Korgaonkar, Patanjali SLPSK, Shankar Balachandran, and Kamakoti Veezhinathan, "The implications of shared data synchronization techniques on multi-core energy efficiency," in *Workshop on Power-Aware Computing Systems*. 2012, USENIX Association.

[10] Gilberto Contreras and Margaret Martonosi, "Power prediction for intel xscale processors using performance monitoring unit events," in *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*. 2005, pp. 221–226, ACM.

[11] Shinan Wang, Hui Chen, and Weisong Shi, "Span: A software power analyzer for multicore computer systems," *Sustainable Computing: Informatics and Systems*, vol. 1, no. 1, pp. 23–34, 2011.

[12] Canturk Isci and Margaret Martonosi, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *International Symposium on Microarchitecture (MICRO)*. 2003, pp. 93–104, ACM/IEEE Computer Society.

[13] Vivek Tiwari, Sharad Malik, and Andrew Wolfe, "Power analysis of embedded software: a first step towards software power minimization," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1994, pp. 384–390, IEEE Computer Society / ACM.

[14] Rong Ge and Kirk W. Cameron, "Power-aware speedup," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. 2007, pp. 1–10, IEEE.

[15] Jee W. Choi, Marat Dukhan, Xing Liu, and Richard W. Vuduc, "Algorithmic time, energy, and power on candidate HPC compute building blocks," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. 2014, pp. 447–457, IEEE Computer Society.

[16] Håkan Sundell and Philippas Tsigas, "NOBLE: non-blocking programming support via lock-free shared abstract data types," *SIGARCH Computer Architecture News*, vol. 36, no. 5, pp. 80–87, 2008.

[17] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le, "RAPL: memory power estimation and capping," in *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*. 2010, pp. 189–194, ACM.

[18] Shirley Browne, Jack J. Dongarra, Nathan Garner, George Ho, and Philip Mucci, "A portable programming interface for performance evaluation on modern processors," *International Journal of High Performance Computing Applications (IJHPCA)*, vol. 14, no. 3, pp. 189–204, 2000.

[19] Vincent M. Weaver, Matt Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Daniel Terpstra, and Shirley Moore, "Measuring energy and power with PAPI," in *Proceedings of International Conference on Parallel Processing Workshops (ICPPW)*. 2012, pp. 262–268, IEEE Computer Society.

[20] Aras Atalar, Anders Gidenstam, Paul Renaud-Goud, and Philippas Tsigas, "Modeling energy consumption of lock-free queue implementations," Tech. Rep. 2014:15, Chalmers University of Technology, 2014.

[21] Håkan Sundell, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas, "A lock-free algorithm for concurrent bags," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2011, pp. 335–344, ACM.

[22] Benoit B. Mandelbrot, "Fractal aspects of the iteration of $z \rightarrow \lambda z(1-z)$ for complex λ and z," *Annals of the New York Academy of Sciences*, vol. 357, pp. 249–259, 1980.

[23] Gabriele Paoloni, "How to benchmark code execution times on Intel® ia-32 and ia-64 instruction set architectures," Tech. Rep. 324264-001, Intel, 2010.

RESULT IV

Aras Atalar, Paul Renaud-Goud and Philippas Tsigas

Lock-Free Search Data Structures: Throughput Modeling with Poisson Processes

Under Submission

5

RESULT IV - Lock-Free Search Data Structures: Throughput Modeling with Poisson Processes

Abstract

This paper considers the modeling and the analysis of the performance of lock-free concurrent search data structures. Our analysis considers such lock-free data structures that are utilized through a sequence of operations which are generated with a memoryless and stationary access pattern. Our main contribution is a new way of analysing lock-free search data structures: our execution model matches with the behavior that we observe in practice and achieves

good throughput predictions. Search data structures are formed of linked basic blocks, usually referred as nodes, that can be accessed by two kinds of events, characterized by their latencies; (i) *CAS* events originated as a result of modifications of the search data structures (ii) *Read* events originated during traversals. This type of data structures are usually designed to accommodate a large number of data nodes, which makes the occurrence of an event on a given node rare at any given time. The throughput is defined by the number of events per operation in conjunction with the factors that impact the latencies of these events. We frame these impacting factors under capacity and coherence cache misses.

In this context, we model the events as Poisson processes that we can merge and split to estimate the latencies of the events based on the interleaving of events from different threads, and in turn estimate the throughput. We have validated our analysis on several fundamental lock-free search data structures such as linked lists, hash tables, skip lists and binary trees.

5.1 Introduction

A search data structure is a collection of $\langle key, value \rangle$ pairs which are stored in an organized way to allow efficient search, delete and insert operations. Linked lists, hash tables, binary trees are some widely known examples. Lock-free implementations of such concurrent data structures are known to be strongly competitive at tackling scalability by allowing processors to operate asynchronously on the data structure.

Performance (here throughput, *i.e.* number of operations per unit of time) is ruled by the number of events in a search data structure operation (*e.g.* $O(\log \mathcal{N})$ for the expected number of steps in a skip list or a binary tree). The practical performance estimation requires an additional layer as the cost (latency) of these events need to be mapped onto the hardware platform; typical values of latency varies from 4 cycles for an access to the first level of cache, to 350 cycles for the last level of remote cache. To estimate the latency of events, one needs to consider the misses, which are sensitive to the interleaving of these events on

the time line. On the one hand, a capacity miss in data or TLB (Translation Lookaside Buffer) caches with LRU (Least Recently Used) policy arise when the interleaving of memory accesses evicted a cacheline. On the other hand, the coherence cache misses arise as a result of the modifications, that are often realized with *Compare-and-Swap* (*CAS*) instructions, in the lock-free search data structure. The interleaving of events that originate from different threads, determine the frequency and severity of these misses, hence the latencies of the events.

In the literature, there exist many asymptotic analyses on the time complexity of sequential search data structures and amortized analyses for the concurrent lock-free variants that involve the interaction between multiple threads. But they only consider the number of events, ignoring the latency. On the other side, there are performance analyses that aim to estimate the coherence and capacity misses for the programs on a given platform, with no view on data structures. We will mention them in the related work. However, there is a lack of results that merge these approaches in the context of lock-free data structures to analytically predict the practical performance.

An analytical performance prediction framework could be useful in many ways: (i) to facilitate design decisions by providing an extensive understanding; (ii) to rank different designs in various contexts; (iii) to help the tuning process. On this last point, lock-free data structures come with specific parameters, *e.g.* padding, back-off and memory management related parameters, and become competitive only after picking their hopefully optimal values.

In this paper, we aim to compute the *average* throughput of search data structures for a sequence of operations, generated by a memoryless and stationary access pattern. The threads execute the same piece of code on the same platform, throughput $\mathcal{T}$ can be estimated on the long-term as the number of threads P divided by the expected latency of an operation (subjected to the distribution of the operations). As the traversal of a search data structure is light in computation, the latency of an operation is dominated by the memory access costs to the nodes that belong to the path from the entry of the data structure to the targeted node.

Therefore, part of this paper is dedicated to the discovery of the route(s) followed by a thread on its way to reach any node in the data structure. In other words, what is the sequence of nodes that are accessed when a given node is targeted by an operation.

As the latency of an operation is the sum of the latency of each memory access to the nodes that are on the path, we obviously need to estimate the individual latency of each traversed node. Even if, in the end, we are interested in the average throughput, this part of the analysis cannot be satisfied with a high-level approach, where we would ignore which thread accesses which node across time. For instance, the cache, whose misses are expected to greatly impact throughput, should be taken carefully into account. This can only be done in a framework from which the interleaving of memory accesses among threads can be extracted. That is why we model the distribution of the memory accesses for every thread.

More precisely, a memory access (*traversal*) can be either the read or the modification of a node, and two point distributions per node represent the triggering instant of either a *Read* or a *CAS*. These point distributions are modeled as Poisson point processes, since they can be approximated by Bernoulli processes, in the context of rare events. Knowing the probabilistic ordering of these events gives a decisive information that is used in the estimate of the traversal latency associated with the triggered event. Once this information is grabbed, we roll back to the expectation of the traversal of a node, then to the expectation of the latency of an operation.

We validate our approach through a large set of experiments on several lock-free search data structures based on various algorithmic designs, namely linked lists, hash tables, skip lists and binary trees. We feed our experiments with different key distributions, and show that our framework is able to predict and explain the observed phenomena.

The rest of the paper is organized as follows. We discuss related work in Section 5.2, then the problem is formulated in Section 5.3. We present the framework in Section 5.4 and the computation of throughput in Section 5.5. In Section 5.6, we show how to initiate our model by considering the particularity

of different search data structures. Finally, we describe the experimental results in Sections 5.7 and 5.8.

5.2 Related Work

The search path length of skiplists is analyzed in [1, 2]. In [1], the search path length is split into vertical and horizontal components, where the horizontal cost is modeled with the number of right-to-left maximas (which corresponds to the traversed node) in a sequence of nodes with random heights. In [3–5], various performance shapers for the randomized trees are studied, such as the time complexity of operations, the expectation and distribution of the depth of the nodes based on their keys.

Previously mentioned studies are not concerned with the interaction between the algorithms and the hardware. The following approaches rely on the independent reference model (IRM) for memory references and derive theoretical results or performance analysis. In [6], data reuse distance patterns are modeled and then exploited to predict the cache miss ratio. In [7], the exact cache miss ratio is derived analytically (computationally expensive) for LRU caches under IRM. As an outcome of this approach, the cache miss ratio of a static binary tree is estimated by assigning independent reference probabilities to the nodes in [8]. This approach provide satisfactory results and also revealed that the impact of the degree of set-associativity is negligible for the cache miss ratios for this scenario.

For the time complexity of lock-free search data structures, asymptotic amortized analyses [9, 10] are conducted since it is not possible to bound the execution time of a single operation, by definition. Apart from these theoretical studies, the performance of concurrent lock-free search data structures are studied and investigated through empirical studies in [11, 12]. In [13], it is shown experimentally that the conflicts between threads occur very rarely in the context of concurrent search data structures, which is confirmed by our analysis.

Procedure AbstractAlgorithm

```
1 while ! done do
2 |  key ← SelectKey(keyPMF);
3 |  operation ← SelectOperation(operationPMF);
4 |  result ← SearchDataStructure(key, operation);
```

Figure 5.1: Generic framework

5.3 Problem Statement

We describe in this section the structure of the algorithm and the system that is covered by our model. We target a multicore platform where the communication between threads takes place through asynchronous shared memory accesses. The threads are pinned to separate cores and call AbstractAlgorithm (see Figure 5.1) when they are spawned.

A concurrent search data structure is a shared collection of data elements, each associated with a key, that support three basic operations holding a key as a parameter. Search (resp. Insert, Delete) operation returns (resp. inserts, deletes) the element if the associated key is present (resp. absent, present) in the search data structure, otherwise returns *null*.

The applications that use a search data structure can be seen as a sequence of operations on the structure, interleaved by application-specific code containing at least the key and operation selection, as reflected in AbstractAlgorithm.

The access pattern (*i.e.* the output of the key and operation selections) should be considered with care since it plays a decisive role in the throughput value. An application that always looks for the first element of a linked list will obviously lead to very high throughput rates. In this study, we consider a memoryless and stationary key and operation selection process *i.e.* such that the probability of selecting a key (resp. an operation type) is a constant.

A search data structure is modeled as a set of basic blocks called nodes, which either contain a value (*valued nodes*) or routes towards nodes (*router nodes*). W.l.o.g. the key set can be reduced to $[1..\mathcal{R}]$, where $\mathcal{R}$ is the number of possible keys. We denote by $(N_i)_{i\in[1..\mathcal{N}]}$ the set of $\mathcal{N}$ potential nodes, and

by K_i the key associated with N_i. Until further notice (see Section 5.8), we assume that we have exactly one node per cacheline.

An operation can trigger two types of events in a node. We distinguish these events as *Read* and *CAS* events. The latency of an event is based on the state of the hardware platform at the time that the event occurs, *e.g.* the level of the cache where a node belongs to for a *Read* request. We summarize the parameters of our model as follows:

- *Algorithm parameters:* Expected latency of the application specific-code (interleaves data structure operations) t^{app}, expected local computational cost to traverse a node t^{cmp}, probability mass functions for the key and operation selection.

- *Platform parameters:* Cache hit latencies (resp. capacity) from level ℓ: t_ℓ^{dat} (resp. C_ℓ^{dat}) for the data caches and t_ℓ^{tlb} (resp. C_ℓ^{tlb}) for TLB caches; other memory instruction latencies (that depends on P): t^{cas} for a *CAS* execution and t^{rec} to recover from an invalid state (*Read* at an invalid cache line, that is in Modified state s in another threads local cache); number of threads P.

5.4 Framework

5.4.1 Event Distributions

We consider first a single thread running AbstractAlgorithm on a data structure where only search operations happen, and we observe the distribution of the *Read* triggering events on a given node N_i. The execution is composed of a sequence of search operations, where each operation is associated with a set of traversed nodes, which potentially includes N_i. If we slice the time into consecutive intervals, where an interval begins with a call to an operation, we can model the *Read* events as a Bernoulli process (where a success means that a *Read* event on N_i occurs), where the probability of having a *Read* event during an interval depends on the associated (key and type) operation (recall that the operation generating process is stationary and memoryless).

Search data structures have been designed as a way to store large data sets while still being able to reach any node within a short time: the set of traversed nodes is then expected to be small in front of the set of all nodes. This implies that, given an operation, the probability that N_i belongs to the set of traversed nodes is small. Therefore we can map the Bernoulli process on the timeline with constant-sized operation interval of length $\mathcal{T}^{-1}$ instead of mapping it with the actual operation intervals: as the probability of having a *Read* event within an operation is small, the duration between two events is big, and this duration is close to the number of initial intervals within this duration, multiplied by $\mathcal{T}^{-1}$ (with high probability, because of the Central Limit Theorem).

When we increase the scope of the operations to insertion and deletion, the structure is no longer static and the probability for a node to appear in an interval is no longer uniform, since it can move inside the data structure. There exists a long line of research in approximating Bernoulli processes by Poisson point processes [14–16]. In particular, [17] has dealt with non-uniform Bernoulli processes. Their error bounds, which are proportional to the success probability, strengthen the use of Poisson processes in our context: the events on N_i are rare, thus the probabilities in Bernoulli processes are small and the approximation is well-conditioned.

Once the *Read* and *CAS* triggering events are modeled as Poisson processes for a single thread, the merge of several Poisson processes models the multi-thread execution.

Lastly, we specify a point on the dynamicity: since we have insertions and deletions, nodes can enter and leave the data structure. This is modeled by the masking random variable P_i which expresses the presence of N_i in the structure. At a random time, we denote by D the set of nodes that are inside the data structure, and P_i is set to 1 iff $N_i \in D$. We denote by p_i its probability of success ($p_i = \mathbb{P}[P_i = 1]$). Its evaluation will often rely on the probability that the last update operation on key k was an Insert; we denote it by q_k, and

$$q_k = \frac{\mathbb{P}\left[Op = op_k^{ins}\right]}{\mathbb{P}\left[Op = op_k^{ins}\right] + \mathbb{P}\left[Op = op_k^{del}\right]}.$$

Note that the search data structures contain generally several *sentinel nodes*

which define the boundaries of the structure and are never removed from the structure: their presence probability is 1.

For a given node N_i, we denote by λ_i^{trav} (resp. λ_i^{read}, λ_i^{cas}) the rate of the events triggering a traversal (resp. *Read*, *CAS*) of N_i due to one thread, when $N_i \in D$. op_k^{del} (resp. op_k^{ins}, op_k^{src}) stands for a Delete (resp. Insert, Search) on node key k. The probability for the application to select op_k^o, where $o \in \{ins, del, src\}$ is denoted by $\mathbb{P}[Op = op_k^o]$. $op_k^o \rightsquigarrow cas(N_i)$ (resp. $read(N_i)$) means that during the execution of op_k^o, a *CAS* (resp. a *Read*) occurs on N_i. Putting all together, we derive the rate of the triggering events:

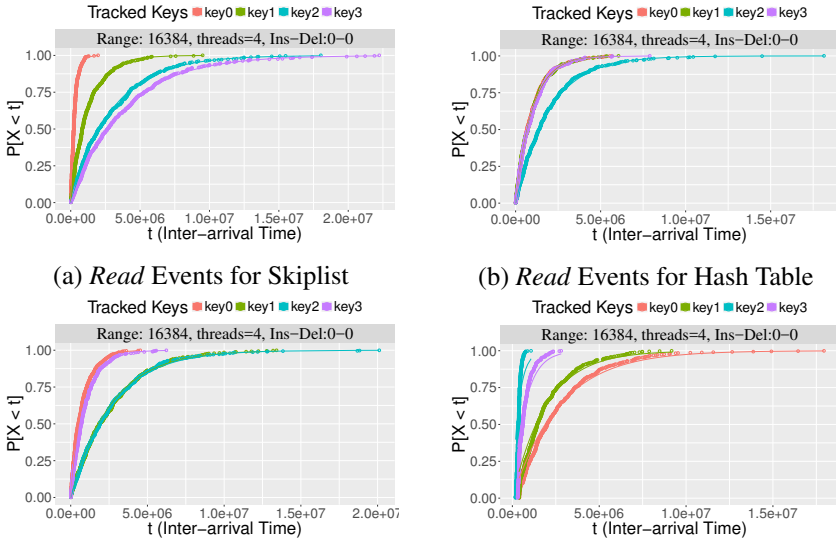
$$\forall e \in \{cas, read\} \ :$$
$$\lambda_i^e = \frac{\mathcal{T}}{P} \times \sum_{o \in \{ins,del,src\}} \sum_{k=1}^{\mathcal{R}} \mathbb{P}[Op = op_k^o] \times \mathbb{P}[op_k^o \rightsquigarrow e(N_i) \mid N_i \in D] \tag{5.1}$$

Recall for later that Poisson processes have useful properties, *e.g.* merging two Poisson processes produces another Poisson process whose rate is the sum of the two initial rates. This implies especially that the traversal triggering events follows a Poisson process with rate $\lambda_i^{trav} = \lambda_i^{read} + \lambda_i^{cas}$, and that the read triggering events that originates from P' different threads and occurs at N_i follow a Poisson process with rate $P' \times \lambda_i^{read}$.

5.4.2 Validity of Poisson Process Hypothesis

To illustrate the validity of modeling the events as Poisson processes, we experimentally extract the cumulative distribution function of the inter-arrival latency of *Read* events that occur on a given node in a skip list and we compare it against the corresponding exponential distribution (recall that the time between events in a Poisson process is exponentially distributed).

We consider a search only scenario and $50/50$ search/update scenario. Each thread initially picks a random key and tracks the instants when a node associated with the chosen key is traversed during the execution. To facilitate the



(a) *Read* Events for Skiplist

(b) *Read* Events for Hash Table

(c) *Read* Events for Binary Tree

(d) *Read* Events for Linked List

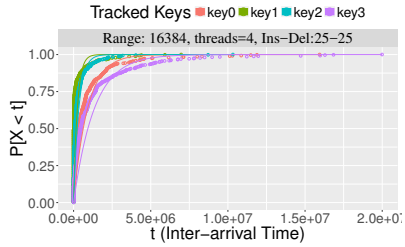
Figure 5.2: Poisson Process Modeling - Search Only

recording of the inter-arrival times, we disable the deletion of these particular keys (deletion is still enabled for any other key).

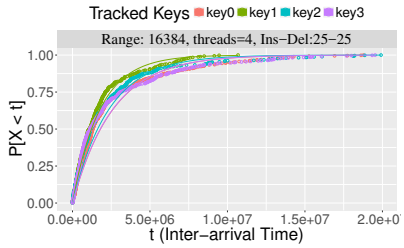
In Figure 5.2 and Figure 5.3, we illustrate the results, where the dots represent the experimental measurements and the lines are generated by exponential distributions. The mean of each distribution is instantiated as the mean of the experimental measurements. One can observe the grounds *a posteriori* of our Poisson process modeling, and the variation of the event rates across keys, issuing from the differences between the node characteristics (key, height, location; see Section 5.6).

5.4.3 Impacting Factors

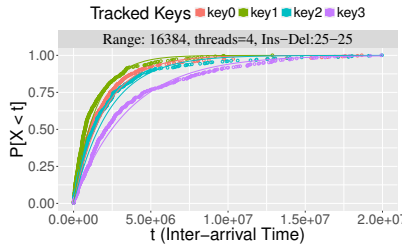
We have identified five factors that dominate the traversal latency of a node, distributed into two sets. On the one hand, the first set of factors only emerges



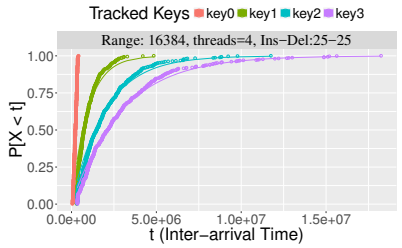
(a) *Read* Events for Skiplist



(b) *Read* Events for Hash Table



(c) *Read* Events for Binary Tree



(d) *Read* Events for Linked List

Figure 5.3: Poisson Process Modeling - 50/50 Search/Update

in the parallel executions as a result of the coherence issues on the search data structures. Atomic primitives, such as a *CAS*, are used to modify the shared search data structures asynchronously. To execute a *CAS* in multi-core architectures, the cache coherency protocol enforces exclusive ownership of the target cacheline by a thread (pinned to a core) through the invalidation of all the other copies of the cacheline in the system, if needed. One can guess the performance implications of this process that triggers back and forth communication among the cores. As the first factor, *CAS* instruction has a significant latency. The thread that executes the *CAS* pays this latency cost. Secondly, any other thread has to stall until the end of the *CAS* execution if it attempts to access (read or modify) the node while the *CAS* is getting executed. Last and most importantly, any thread pays a cost to bring a cacheline to a valid state if it attempts to access a node that resides in this cacheline and that has been modified by another

thread after its previous access to this node.

On the other hand, the capacity misses in the data and TLB caches are other performance impacting factors for the node traversals. Consider a cache of size C (fully associative), assume a node is traversed by a thread at time t and the next traversal (same thread and node) occurs at time t'. The thread would experience a capacity miss for the traversal at time t' if it has traversed at least C distinct nodes in the interval (t, t'). The same applies for TLB caches where the references to the distinct pages are counted instead of the nodes.

At a given instant, we denote by $Traverse_i$ the latency of traversing node N_i, either due to a *Read* event or a *CAS* event, for a given thread. This latency is the sum of random variables that correspond to the previous respective five impacting factors:

$$Traverse_i = CAS_i^{exe} + CAS_i^{stall} + CAS_i^{reco} + \sum_{\ell} Hit_i^{cache_\ell} + \sum_{\ell} Hit_i^{tlb_\ell}, \tag{5.2}$$

where, at a random time, CAS_i^{exe} is the latency of a *CAS*, CAS_i^{stall} the stall time implied by other threads executing a *CAS* on N_i, CAS_i^{reco} the time needed to fetch the data from another modifying thread, $Hit_i^{cache_\ell}$ the latency resulting from a hit on the data cache in level ℓ, and $Hit_i^{tlb_\ell}$ the latency coming from a hit on the TLB cache in level ℓ.

5.4.4 Solving Process

The solving decomposes into three main steps. Firstly, we can notice that Equation 5.1 exposes $2\mathcal{R}+1$ unknowns (the $2\mathcal{R}$ access rates and throughput) against $2\mathcal{R}$ equations. To end up with a unique solution, a last equation is necessary. The first two steps provide a last sufficient equation thanks to Little's law (see Section 5.5.2), which links throughput with the expectation of the traversal latency of a node, computed from Sections 5.5.1.1 to 5.5.1.6. We show in these sections that they can be expressed according to the access rates λ_i^{read} and λ_i^{cas}. The last step focuses on the values of the probabilities in Equation 5.1, which are strongly related with the particular data structure under consideration; they

are instantiated in Section 5.6.1 (resp. 5.6.2, 5.6.3, 5.6.4) for linked lists (resp. hash tables, skip lists, binary trees).

5.5 Throughput Estimation

5.5.1 Traversal Latency

Applying expectation to Equation 5.2 leads to $\mathbb{E}\left[Traverse_i\right] = \mathbb{E}\left[CAS_i^{exe}\right] + \mathbb{E}\left[CAS_i^{stall}\right] + \mathbb{E}\left[CAS_i^{reco}\right] + \mathbb{E}\left[\sum_\ell Hit_i^{cache_\ell}\right] + \mathbb{E}\left[\sum_\ell Hit_i^{tlb_\ell}\right]$. We express here each term according to the rates at every node $\lambda_\star^{cas}$ and $\lambda_\star^{read}$.

5.5.1.1 CAS Execution

Naturally, among all traversal events, only the events originating from a *CAS* event contribute, with the latency t^{cas} of a *CAS*: $\mathbb{E}\left[CAS_i^{exe}\right] = t^{cas} \cdot \lambda_i^{cas} / (\lambda_i^{read} + \lambda_i^{cas})$.

5.5.1.2 Stall Time

A thread experiences stall time while traversing N_i when a thread, among the $(P-1)$ remaining threads, is currently executing a *CAS* on the same node. As a first approximation, supported by the rareness of the events, we assume that at most one thread will wait for the access to the node.

Firstly, we obtain the rate of *CAS* events generated by $(P-1)$ threads through the merge of their poisson processes. Consider a traversal of N_i at a random time; (i) the probability of being stalled is the ratio of time when N_i is occupied by a *CAS* of $(P-1)$ threads, given by: $\lambda_i^{cas}(P-1)t^{cas}$; (ii) the stall time that the thread would experience is distributed uniformly in the interval $[0, t^{cas}]$. Then, we obtain: $\mathbb{E}\left[CAS_i^{stall}\right] = \lambda_i^{cas}(P-1)t^{cas}(t^{cas}/2)$.

5.5.1.3 Invalidation Recovery

Given a thread, a coherence cache miss occurs if N_i is modified by any other thread in between two consecutive traversals of N_i. The events that are con-

cerned are: (i) the *CAS* events from any thread; (ii) the *Read* events from the given thread. When N_i is traversed, we look back at these events, and if among them, the last event was a *CAS* from another thread, a coherence miss occur: $\mathbb{P}\left[\text{Coherence Miss on } N_i\right] = \frac{\lambda_i^{cas}(P-1)}{\lambda_i^{cas}P+\lambda_i^{read}}$. We derive the expected latency of this factor during a traversal at N_k by multiplying this with the latency penalty of a coherence cache miss: $\mathbb{E}\left[CAS_i^{reco}\right] = \mathbb{P}\left[\text{coherence miss on } N_i\right] \times t^{rec}$.

5.5.1.4 Che's Approximation

Che's Approximation is a technique to estimate the hit ratio of a LRU cache, where the object (nodes for our case) accesses follow IRM (Independent Reference Model). Che's approximation is concerned with the capacity misses in a cache. We apply the approximation to the search data structures to estimate $\mathbb{E}\left[Hit_i^{cache_\ell}\right]$ and $\mathbb{E}\left[Hit_i^{tlb_\ell}\right]$. In this part, we give a brief discussion on Che's Approximation and in the following sections (see 5.5.1.5, 5.5.1.6), we have shown how we adapt this scheme for our purposes.

IRM is based on the assumption that the object references occur in an infinite sequence from a fixed catalog of $\mathcal{N}$ objects. The probability of referencing object i at any point in the sequence (denoted by s_i, where $i \in [1..\mathcal{N}]$) is a constant that does not depend on the reference history and does not vary over time. Under LRU policy with cache of size C_ℓ^{dat} and subject to IRM demand of $\mathcal{N}$ objects, an object reference would lead to a capacity miss if at least C_ℓ^{dat} unique object references take place after the previous reference to the same object. Let a reference to object i (O_i) occurs at time t_0, the characteristic time for the object i is defined by the random variable:

$$T_\ell^i = \inf\{t > 0 : X^i(t) = C_\ell^{dat}\}, \text{where,}$$

$$X^i(t) = \sum_{j=1, j\neq i}^{\mathcal{N}} \mathbf{1}_{t_0 < O_j \leq t}$$

Briefly, Che's approximation, first combines all T_ℓ^i, where $i \in [1..\mathcal{N}]$ in a single variable by assuming s_i is negligible compared to $\sum_{j=1}^{\mathcal{N}} s_j$ and then approximates T_ℓ^i with a constant T_ℓ^{dat} over objects. Consider a sequence of references that follows an IRM demand for $\mathcal{N}$ objects, with reference probability s_i, where $i \in [1..\mathcal{N}]$. The characteristic time T_ℓ^{dat} of a cache with size C_ℓ^{dat} is the unique solution of the following equation:

$$C_\ell^{dat} = \sum_{i=1}^{\mathcal{N}} (1 - e^{-s_i T_\ell^{dat}})$$

In [18], they analyze and illustrate the reason behind the accuracy of the approximations for a quite large spectrum of object reference distributions. Their argument relies on the random variable $X(t) = \sum_{j=1}^{\mathcal{N}} \mathbf{1}_{t_0 < O_j \leq t}$, that provides the number of unique object references that have occured in the interval $[0, t]$. As the crucial property, $X(t)$ is defined as the sum of independent random variables. Based on the central limit theorem, they show that a Gaussian approximation for this sum is quite reasonable, for all t.

Without loss of generality, let an object i is referenced consecutively at time 0 and t. We know that the second reference would be cache miss, in a cache of size C_ℓ^{dat}, if $X(t) > C_\ell^{dat}$, where by assumption $X(t)$ is a Gaussion random variable. The cache hit ratio of cacheline is given by:

$$hit_\ell^i = 1 - \int_0^{+\infty} \mathbb{P}\left[X(t) > C_\ell^{dat}\right] s_i e^{-s_i t} dt \tag{5.3}$$

Che's approximation, basically, approximates the cumulative distribution function of $X(t)$ with a step function that cuts this S-shaped cumulative distribution function at the $\mathbb{E}[X(t)] = \sum_{i=1}^{\mathcal{N}} (1 - e^{-s_i t})$, denoted by $m(t)$. Thus, it approximates hit_ℓ^i in Equation 5.3 with:

$$\begin{aligned} hit_\ell^i &\approx 1 - \int_0^{+\infty} \mathbf{1}_{m(t) > C_\ell^{dat}} s_i e^{-s_i t} dt \\ &= 1 - \int_0^{+\infty} \mathbf{1}_{t > T_\ell^{dat}} s_i e^{-s_i t} dt \end{aligned}$$

In this study, we have exploited Che's approximation to estimate the data and TLB cache hit ratios with a slight modification by keeping our arguments along the same lines with the ones presented above.

5.5.1.5 Cache Misses

We consider a data cache at level ℓ of size C_ℓ^{dat} and compute the hit latency due to *Read* events on this cache. We assume that N_i is either present in the search data structure or not, during the characteristic time of the cache. *Read* events at N_i are indeed much more frequent than the removal or insertion of N_i. This implies that if the characteristic time is long enough to accommodate the intervals where $N_i \in D$ and $N_i \notin D$, then the cache miss ratio of N_i should be quite low, which would be underestimated due to our assumption. We can employ the *Read* rates as popularities, *i.e.* $s_i = \lambda_i^{read}$, and modify Che's approximation to discriminate whether, at a random time, N_i is inside the data structure or not.

We integrate the masking variable P_i into Che's approximation. We have: $X^{cache}(t) = \sum_{i=1}^{\mathcal{N}} P_i \mathbf{1}_{0<O_i\leq t}$, where O_i denotes the reference time of N_i. We can still assume $X^{cache}(t)$ is gaussian, as a sum of many independent random variables. We estimate the characteristic time as follows with the linearity of expectation and the independence of the random variables:

$$\begin{aligned}\mathbb{E}\left[X^{cache}(t)\right] &= \sum_{i=1}^{\mathcal{N}} \mathbb{E}\left[P_i \mathbf{1}_{0<O_i\leq t}\right] = \sum_{i=1}^{\mathcal{N}} \mathbb{E}\left[P_i\right] \mathbb{E}\left[\mathbf{1}_{0<O_i\leq t}\right] \\ &= \sum_{i=1}^{\mathcal{N}} p_i(1 - e^{-\lambda_i^{read} t}).\end{aligned}$$

Lastly, we solve the equation for the characteristic time T_ℓ^{dat} of level ℓ cache: $\sum_{i=1}^{\mathcal{N}} p_i(1 - e^{-\lambda_i^{read} T_\ell^{dat}}) = C_\ell^{dat}$ thanks to a fixed-point approach. After computing T_ℓ^{dat}, we estimate the cache hit ratio (on level ℓ) of N_i: $1 - e^{-\lambda_i^{read} T_\ell^{dat}}$.

5.5.1.6 Page Misses

In this paragraph, we aim at computing the page hit ratio of N_i for the TLB cache at level ℓ of size C_ℓ^{tlb}. The total number $\mathcal{M}$ of pages that are used by the search data structure can be regulated by a parameter of the memory managements scheme (frequency of recycling attempts for the deleted nodes), as the total number of nodes is a function of $\mathcal{R}$. Different from the cachelines (corresponding to the nodes), we can safely assume that a page accommodates at least a single node that is present in the structure at any time.

We cannot apply straightforwardly Che's approximation since the page reference probabilities are unknown. However, we are given the cacheline reference probabilities $s_i = \lambda_i^{read}$ for $i \in [1..\mathcal{N}]$ and we assume that $\mathcal{N}$ cachelines are mapped uniformly to $\mathcal{M}$ pages, $[1..\mathcal{N}] \to [1..\mathcal{M}]$, $\mathcal{N} > \mathcal{M}$. Under these assumptions, we know that the resulting page references would follow IRM because aggregated Poisson processes form again a poisson process.

We follow the same line of reasoning as in the cache miss estimation. First, we consider a set of Bernoulli random variables (Y_i^j), leading to a success if N_i is mapped into page j, with probability $p_i/\mathcal{M}$ (hence Y_i^j does not depend on j). Under IRM, we can then express the page references as point processes with rate $r_j = \sum_{i=1}^{\mathcal{N}} Y_i^j s_i$, for all $j \in [1..\mathcal{M}]$.

Similar to the previous section, we denote the time of a reference to page j with O_j and we define the random variable $X^{page}(t) = \sum_{j=1}^{\mathcal{M}} \mathbf{1}_{0<O_j \leq t}$ and

compute its expectation:

$$\begin{aligned}
\mathbb{E}\left[X^{page}(t)\right] &= \sum_{j=1}^{\mathcal{M}} \mathbb{E}\left[\mathbf{1}_{0<O_j\leq t}\right] = \sum_{j=1}^{\mathcal{M}} \mathbb{E}\left[1-e^{-r_j t}\right] \\
&= \sum_{j=1}^{\mathcal{M}} \mathbb{E}\left[1-e^{-\sum_{i=1}^{\mathcal{N}} Y_i^j \lambda_i^{read} t}\right] \\
&= \sum_{j=1}^{\mathcal{M}} \left(1-\prod_{i=1}^{\mathcal{N}} \mathbb{E}\left[e^{-Y_i^j \lambda_i^{read} t}\right]\right) \\
&= \sum_{j=1}^{\mathcal{M}} \left(1-\prod_{i=1}^{\mathcal{N}} \left(\frac{\mathcal{M}-p_i}{\mathcal{M}} + \frac{p_i e^{-\lambda_i^{read} t}}{\mathcal{M}}\right)\right) \\
&= \mathcal{M}\left(1-\prod_{i=1}^{\mathcal{N}} \left(\frac{\mathcal{M}-p_i}{\mathcal{M}} + \frac{p_i e^{-\lambda_i^{read} t}}{\mathcal{M}}\right)\right),
\end{aligned}$$

Assuming $X^{page}(t)$ is Gaussian as it is sum of many independent random variables, we solve the following equation for the constant T_ℓ^{tlb} (characteristic time of a TLB cache of size C): $\mathbb{E}\left[X^{page}(T_\ell^{tlb})\right] = C_\ell^{tlb}$.

Lastly, we obtain the TLB hit rate for N_i by relying on the average *Read* rate of the page that N_i belongs to; we should add to the contributions of N_i, the references to of the nodes that belong to the same page as N_i. Then follows the TLB hit ratio: $1-e^{-z_i T_\ell^{tlb}}$, where

$$z_i = \lambda_i^{read} + \mathbb{E}\left[\sum_{j=1, j\neq i}^{\mathcal{N}} Y_j^k \lambda_j^{read}\right] = \lambda_i^{read} + \sum_{j=1, j\neq i}^{\mathcal{N}} p_j \lambda_j^{read}/\mathcal{M}.$$

5.5.1.7 Interactions

To be complete, we mention the interaction between impacting factors and the possibility of latency overlaps in the pipeline. Firstly, the traversal latency of different nodes cannot overlap due to the semantic dependency for the linked nodes. For a single node traversal, the latency for cas execution and stall time cannot overlap with any other factor. We consider inclusive data and TLB caches. It is not possible to have a cache hit on level l, if the

cache on level $l-1$ is hit, and we do not consider any cost for the data cache hit if invalidation recovery (coherence) cost is induced (*i.e.* $\mathbb{E}\left[Hit_i^{cache_\ell}\right] = (1-\mathbb{P}\left[coherence\ miss\right])(\mathbb{P}\left[hit\ cache_l\right]-\mathbb{P}\left[hit\ cache_{l-1}\right])t_\ell^{dat}$).

5.5.2 Latency vs. Throughput

In the previous sections, we have shown how to compute the expected traversal latency for a given node. There remains to combine these traversal latencies in order to obtain the throughput of the search data structure. Given $N_i \in D$, the average arrival rate of threads to N_i is $\lambda_i^{trav} = \lambda_i^{read} + \lambda_i^{cas}$. Thus the average arrival rate of threads to N_i is: $p_i\lambda_i^{trav}$. It can then be passed to Little's Law [19], which states that the expected number of threads (denoted by t_i) traversing N_i obeys to $t_i = p_i\lambda_i^{trav}\mathbb{E}\left[Traverse_i\right]$. The equation holds for any node in the search data structure, and for the application call occurring in between search data structure operations. Its expected latency is a parameter ($\mathbb{E}\left[Traverse_0\right] = t^{app}$) and its average arrival rate is equal to the throughput ($\lambda_0^{trav} = \mathcal{T}$). Then, we have: $\sum_{i=0}^{\mathcal{N}} t_i = \sum_{i=0}^{\mathcal{N}}(p_i\lambda_i^{trav}\mathbb{E}\left[Traverse_i\right])$, where λ_i^{trav} and $\mathbb{E}\left[Traverse_i\right]$ are linear functions of $\mathcal{T}$. We also know $\sum_{i=0}^{\mathcal{N}} t_i = P$ as the threads should be executing some component of the program. We define constants with a_i, b_i, c_i for $i \in [0..\mathcal{N}]$. And, we represent $\lambda_i^{trav} = a_i\mathcal{T}$ and $\mathbb{E}\left[Traverse_i\right] = b_i\mathcal{T} + c_i$ and we obtain the following second order equation: $\sum_{i=0}^{\mathcal{N}}(p_i a_i b_i)\mathcal{T}^2 + \sum_{i=0}^{\mathcal{N}}(p_i a_i c_i)\mathcal{T} - P = 0$. This second order equation has a unique positive solution that provides the expected throughput, $\mathcal{T}$.

5.6 Instantiating the Throughput Model

In this section, we show how to initialize our model with widely known lock-free search data structures, that have different operation time complexities. In order to obtain a throughput estimate for a structure, we need to compute the rates $\lambda_\star^{read}$ and $\lambda_\star^{cas}$, and $\mathbb{P}\left[op_k^o \rightsquigarrow e(N_i)\,|N_i \in D\right]$, *i.e.* the probability that, at a random time, an operation of type o on key k leads to a memory instruction of type e on node N_i, knowing that N_i is in the data structure. For the ease

of notation, nodes will sometimes be doubly or triply indexed, and when the context is clear, we will omit $|N_i \in D$ in the probabilities.

We first estimate the throughput of linked lists and hash tables, on which we can directly apply our method, then we move on more involved search data structure, namely skip lists and binary trees, that need a particular attention.

5.6.1 Linked List

We start with the lock-free linked list implementation of Harris [20]. All operations in the linked list start with the search phase in which the linked list is traversed until a key. At this point all operations terminate except the successful update operations that proceed by modifying a subset of nodes in the structure with *CAS* instructions. The structure contains only valued node and two sentinel nodes N_0 and $N_{\mathcal{R}+1}$, so that $\mathcal{N} = \mathcal{R} + 2$ and for all $i \in [1..\mathcal{R}]$, N_i holds key i, *i.e.* $K_i = i$.

First, we need to compute the probabilities of triggering a *Read* event and *CAS* event on a node, given that the node is in the search data structure, for all operations of type $t \in \{\mathsf{Insert}, \mathsf{Delete}, \mathsf{Search}\}$ targeted to key k.

At a random time, N_k, for $k \in [1..\mathcal{R}]$, is in the linked list iff the last update operation on key k is an insert: $p_k = q_k$, by definition of q_k. Moreover, when N_k is in the structure (condition that we omit in the notation), $op^t_{k'}$ reads N_k, either if N_k is before $N_{k'}$, or if it is just after $N_{k'}$. Formally, $\mathbb{P}\left[op^o_{k'} \rightsquigarrow read(N_k)\right] = 1$ if $k \leq k'$ and $\mathbb{P}\left[op^o_{k'} \rightsquigarrow read(N_k)\right] = \prod_{i=k'}^{k-1}(1-p_i)$ if $k > k'$.

CAS events can only be triggered by successful Insert and Delete operations. A successful Insert operation, targeted to $N_{k'}$, is realized with a *CAS* that is executed on N_k, where $k = \sup\{\ell < k' : N_\ell \in D\}$. The probability of success, which conditions the *CAS*'s, follows from the presence probabilities:

$$\mathbb{P}\left[op^{ins}_{k'} \rightsquigarrow cas(N_k)\right] = \begin{cases} 0, & \text{if } k \geq k' \\ \prod_{i=k+1}^{k'} (1-p_i), & \text{if } k < k' \end{cases}$$

$$\mathbb{P}\left[op^{del}_{k'} \rightsquigarrow cas(N_k)\right] = \begin{cases} 1, & \text{if } k = k' \\ 0, & \text{if } k > k' \\ p_{k'} \prod_{i=k+1}^{k'-1} (1 - p_i), & \text{if } k < k' \end{cases}$$

5.6.2 Hash Table

We analyze here a chaining based hash table where elements are hashed to B buckets implemented with the lock-free linked list of Harris [20]. The structure is parametrized with a load factor lf which determines B through $B = \mathcal{R}/lf$. The hash function $h : k \mapsto \lceil k/lf \rceil$ maps the keys sequentially to the buckets, so that, after including the sentinel nodes (2 per bucket), we can doubly index the nodes: $N_{b,k}$ is the node in bucket b with key k, where $b \in [1..B]$ and $k \in [1..lf]$ (the last bucket may contain less elements).

$$\mathbb{P}\left[op^{o}_{b',k'} \rightsquigarrow read\big(N_{b,k}\big)\right] = \begin{cases} 0, & \text{if } b' \neq b \\ 1, & \text{if } b' = b \text{ and } k' \geq k \\ \prod_{j=k'}^{k-1} (1 - p_{b,j}), & \text{if } b' = b \text{ and } k' < k \end{cases}$$

$$\mathbb{P}\left[op^{ins}_{b',k'} \rightsquigarrow cas\big(N_{b,k}\big)\right] = \begin{cases} 0, & \text{if } b' \neq b \text{ or } k' \leq k \\ \prod_{j=k+1}^{k'} (1 - p_{b,j}), & \text{if } b' = b \text{ and } k' > k \end{cases}$$

$$\mathbb{P}\left[op^{del}_{b',k'} \rightsquigarrow cas\big(N_{b,k}\big)\right] = \begin{cases} 0, & \text{if } b' \neq b \text{ or } k' < k \\ 1, & \text{if } b' = b \text{ and } k' = k \\ p_{b,k'} \prod_{j=k+1}^{k'-1} (1 - p_{b,j}), & \text{if } b' = b \text{ and } k' > k \end{cases}$$

In the previous two data structures, we do observe differences in the traversal rate from node to node, but the node associated with a given key does not show significant variation in its traversal rate during the course of the execution: inside the structure, the number of nodes preceding (and following) this node

is indeed rather stable. In the next two data structures, node traversal rates can change dramatically according to node characteristics, that may include its position in the structure. In a skip list, a node N_i containing key K_i with maximum height will be traversed by any operation targeting a node with a higher key. However, N_i can later be deleted and inserted back with the minimum height; the operations that traverse it will then be extremely rare. The same reasoning holds when comparing an internal node with key K_i of a binary tree located at the root or close to the leaves.

As explained before, an accurate cache miss analysis cannot be satisfied with average access rates. Therefore, the information on the possible significant variations of rates should not be diluted into a single access rate of the node. To avoid that, we pass the information through virtual nodes: a node of the structure is divided into a set of virtual nodes, each of them holding a different flavor of the initial node (height of the node in the skip list or subtree size in the binary tree). The virtual nodes go through the whole analysis instead of the initial nodes, before we extract the average behavior of the system hence throughput.

5.6.3 Skip List

There exist various lock-free skip list implementations and we study here the lock-free skip list [21]. Skip lists offer layers of linked lists. Each layer is a sparser version of the layer below where the bottom layer is a linked list that includes all the elements that are present in the search data structure. An element that is present in the layer at height h appears in layer at height $h + 1$ with a fixed appearance probability ($1/2$ for our case) up to some maximum layer h_{max} that is a parameter of the skip list.

Skip list implementations are often realized by distinguishing two type of nodes: (i) valued nodes reside at the bottom layer and they hold the key-value pair in addition to the two pointers, one to the next node at the bottom layer and one to the corresponding routing node (could be *null*); (ii) routing nodes are used to route the threads towards the search key. Being coupled with a valued

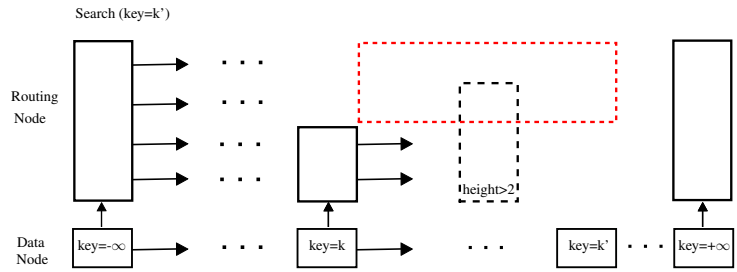


Figure 5.4: Skip List Events: Read Event Probability

node, a routing node does not replicate the key-value pair. Instead, only a set of pointers, corresponding to the valued node containing the next key in different layers, are packed together in a single routing node (that fits in a cacheline with high probability). Every *Read* event in a routing node is preceded by a *Read* in the corresponding valued node.

We denote by $N^{rou}_{k,h}$ the routing node containing key k, whose set of pointers is of height h, where $h \in [1..h_{max}]$. A valued node containing the key k is denoted by $N^{dat}_{k,h}$ when connected to $N^{rou}_{k,h}$ ($h = 0$ if there is no routing node). Furthermore, there are four sentinel nodes $N^{dat}_{0,h_{max}}$, $N^{rou}_{0,h_{max}}$, $N^{dat}_{\mathcal{R}+1,h_{max}}$, $N^{rou}_{\mathcal{R}+1,h_{max}}$. The presence probabilities result from the coin flips (bounded by h_{max}): for $z \in \{dat, rou\}$, $p^z_{k,h} = 2^{-(h+1)}q_k$ if $h < h_{max}$, $p^z_{k,h} = q_k - \sum_{\ell=0}^{h_{max}-1} p^z_{k,\ell}$ otherwise.

By decomposing into three cases, we compute the probability that an operation $op^o_{k'}$ of type $o \in \{ins, del, src\}$, targeted to k', causes a *Read* triggering event at $N^z_{k,h}$ when $N^z_{k,h} \in D$. Let assume first that $k' > k$. The operation triggers a *Read* event at node $N^z_{k,h}$ if for all (x, y) such that $y > h$ and $k < x \leq k'$, $N^z_{x,y}$ is not present in the skip list (*i.e.* in Figure 5.4, no node in the skip list overlaps with the red frame). Let assume now $k' < k$. The occurrence of a *Read* event requires that: for all (x, y) such that $y \geq h$ and $k' \leq x < k$, $N^z_{x,y}$, is not present in the structure. Lastly, a *Read* event is certainly triggered if $k' = k$.

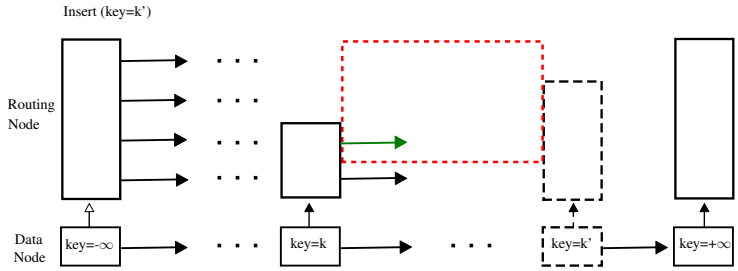


Figure 5.5: Skiplist Events: *CAS* Event Probability

The final formula is given by:

$$\mathbb{P}\left[op^{o}_{k'} \rightsquigarrow read\left(N^{z}_{k,h}\right)\right] = \begin{cases} \prod_{x=k+1}^{k'} \left(1 - \left(\sum_{y=h+1}^{h_{max}} p^{z}_{x,y}\right)\right), & \text{if } k \leq k' \\ \prod_{x=k'}^{k-1} \left(1 - \left(\sum_{y=h}^{h_{max}} p^{z}_{x,y}\right)\right), & \text{if } k > k' \end{cases}$$

Next, we apply a similar approach for *CAS* events. In Figure 5.5, we illustrate an example. A *CAS* event occurs at the green pointer, as a result of the removal (or insertion) of K_k if there is no node in the red frame. For all node and operation couples, $\mathbb{P}\left[op^{o}_{k'} \rightsquigarrow cas\left(N^{z}_{k,h}\right)\right]$ is simply obtained in those lines.

The insertion of an element with $K_{k'}$ introduces $N^{z}_{k',h}$ with probability $2^{-(h+1)}$ if $h \in [1..h_{max}-1]$, and $1 - \sum_{i=0}^{h_{max}-1} 2^{-(h+1)}$ when the maximum height. The data node is linked to the list at the bottom layer with a *CAS* that is executed on the previous data node. If a routing node is introduced, it is linked to lists at h different layers, thus leads to h *CAS* instructions that are applied on the other nodes.

The deletion of an element is composed of two phases. The first phase is to mark the data node, $N^{dat}_{k',h}$ and the pointers in the routing node with height k', if it exists. If the height of the routing node is more than one, it is possible that multiple *CAS* intructions are executed on the same routing node. But, we only consider the first one. The latency and also the effect of remaining ones would be negligible, as they are applied on the same cacheline one after each

other. This repetitive behavior guarentees that the cacheline has already been exclusively owned before the next *CAS* instructions run. To recall, this is consistent with our assumption that an event can occur at most once per operation on a node. The second phase of deletion operation follows the same path with the insertion operation. Simply, a *CAS*, on the previous node, is executed for each layer that the data and routing nodes span.

We have denoted the success probability of an Insert operation with $q_{k'} = \frac{\mathbb{P}\left[op=op_{k'}^{\mathsf{Insert}}\right]}{\mathbb{P}\left[op=op_{k'}^{\mathsf{Insert}}\right]+\mathbb{P}\left[op=op_{k'}^{\mathsf{Delete}}\right]}$. Also, the factor $2^{-(h+1)}$ provides the probability of the insertion of a routing node with height h, coupled with its data node. Based on the non-existence of any node that overlaps with the area that is enclosed with the red frame in Figure 5.5, we obtain:

$$\mathbb{P}\left[op_{k'}^{ins} \rightsquigarrow cas\big(N_{k,h}^{z}\big)\right] =$$
$$\begin{cases} (1-q_{k'})(\sum_{h=0}^{h_{max}} 2^{-(h+1)}(\prod_{x=k+1}^{k'-1}(1-(\sum_{y=h}^{h_{max}} p_{x,y}^{z})))), & \text{if } k < k' \\ 0, & \text{if } k \geq k' \end{cases}$$

$$\mathbb{P}\left[op_{k'}^{del} \rightsquigarrow cas\big(N_{k,h}^{z}\big)\right] =$$
$$\begin{cases} 1, & \text{if } k = k' \\ q_{k'}(\sum_{h=0}^{h_{max}} 2^{-(h+1)}(\prod_{x=i+1}^{k'-1}(1-(\sum_{y=h}^{h_{max}} p_{x,y}^{z})))), & \text{if } k < k' \\ 0, & \text{if } k > k' \end{cases}$$

5.6.4 Binary Tree

We show here how to estimate the throughput of external binary trees. They are composed of two types of nodes: internal nodes route the search towards the leaves (routing nodes) and store just a key, while leaves, referred as external nodes contain the key-value pair (valued node). We use the external binary tree of Natarajan [22] to initialize our model. The search traversal starts and continues with a set of internal nodes and ends with an external node. We denote by N_k^{int} (resp. N_k^{ext}) the internal (resp. external) node containing key

k, where $k \in [1..\mathcal{R}]$. The tree contains two sentinel internal nodes that reside at the top of the tree (hence are traversed by all operation): N_{-1}^{int} and N_0^{int}.

Our first aim is to find the paths followed by any operation through the binary tree, in order to obtain the access triggering rates, thanks to Equation 5.1. Binary trees are more complex than the previous structures since the order of the operations impact the positioning of the nodes. The random permutation model proposes a framework for randomized constructions in which we can develop our model. Each key is associated with a priority, which determines its insertion order: the key with the highest priority is inserted first. The performance characteristics of the randomized binary trees are studied in [4]. In the same vein, we compute the traversal probability of the internal node with key k in an operation that targets key k'.

Lemma 18. *Given an external binary tree, the probability of traversing N_k^{int} in an operation that targets key $K_{k'}$ is given by: (i) $1/f(k,k')$ if $k' \geq k$; (ii) $1/(f(k',k)-1)$ if $k' < k$, where $f(x,y)$ provides the number internal nodes whose keys are in the interval $[x,y]$.*

Proof. N_k^{int} would be traversed if it is on the search path to the external node with key k'. Given $k' \geq k$, this happens iff N_k^{int} has the highest priority among the internal nodes in the interval $[k,k']$. This interval contains $f(k,k')$ internal nodes, thus, the probability of N_k^{int} to possess the highest priority is $1/f(k,k')$. Similarly, if $k' < k$, then N_k^{int} is traversed iff it has the highest priority in the interval $(k',k]$. Hence, the lemma. □

Even if in the binary tree, nodes are inserted and deleted an infinite number of times, Lemma 18 can still be of use. The number of internal nodes in the interval $[k,k']$ (or $(k',k]$ if $k' < k$) is indeed a random variable which is the sum of independent Bernoulli random variables that models the presence of the nodes. As a sum of many independent Bernoulli variables, the outcome is expected to have low variations because of its asymptotic normality. Therefore, we replace this random variable with its expected value and stick to this approximation in the rest of this section. The number of internal nodes in any interval come out from the presence probabilities: $p_k^z = q_k$, where $z \in \{int, ext\}$.

In an operation is targeted to key k', a single external node is traversed (if any): $N^{ext}_{k'}$, if present, else the external node with the biggest key smaller than k', if it exists, else the external node with the smallest key. Then, we have:

$$\mathbb{P}\left[op^o_{k'} \rightsquigarrow read\big(N^{int}_k\big)\right] = \begin{cases} 1/(1+\sum_{i=k'+1}^{k-1} p^{int}_i), \text{ if } k > k' \\ 1/(1+\sum_{i=k+1}^{k'} p^{int}_i), \text{ if } k \leq k' \end{cases},$$

$$\mathbb{P}\left[op^o_{k'} \rightsquigarrow read\big(N^{ext}_k\big)\right] = \begin{cases} 1, & \text{if } k = k' \\ \prod_{i=k+1}^{k'}(1-p^{ext}_i), & \text{if } k < k' \\ \prod_{i=1}^{k-1}(1-p^{ext}_i), & \text{if } k > k' \end{cases}$$

These probabilities finally lead to the computation of the *Read* (resp. *CAS*) rates $\lambda^{read}_{z,k}$ (resp. $\lambda^{cas}_{z,k}$) of N^z_k, where $z \in \{int, ext\}$, that will be used in the last following step.

We focus now on the *Read* rate of the internal nodes. We have found the average behavior of each node in the previous step; however, the node can follow different behaviors during the execution since the *Read* rate of N^{int}_k depends on the size of the subtree whose root is N^{int}_k, which is expected to vary with the update operations on the tree. We dig more into this and reflect these variations by decomposing N^{int}_k into H_k virtual nodes, $N^{int}_{k,h}$, where $h \in [1..H_k]$. We define the *Read* rate $\lambda^{read}_{int,k,h}$ of these virtual nodes as a weighted sum of the initial node rate thanks the two equations $p^{int}_k = \sum_{h=1}^{H_k} p^{int}_{k,h}$ and $p^{int}_k \lambda^{read}_{int,k} = \sum_{h=1}^{H_k} p^{int}_{k,h} \lambda^{read}_{int,k,h}$.

We connect the virtual nodes to the initial nodes in two ways. On the one hand, one can remark that the *Read* rate is proportional to the subtree size: $\lambda^{read}_{int,k,h} \propto h\lambda^{read}_{int,k}$. On the other hand, based on the probability mass function of the random variable Sub_k representing the size of the subtree rooted at N^{int}_k, we can evaluate the weight of the virtual nodes: $p^{int}_{k,h} = p^{int}_k \mathbb{P}\left[Sub_k = h\right]$.

We have computed $\lambda^{read}_{int,k}$. These values reflect the average behaviour along the whole execution. However, the average behavior is not enough to computethe traversal latency accurately for the internal nodes. In the execution, there are different time intervals where $\lambda^{read}_{int,k}$ show significant variation depending on the part of the tree that it is located. For instance, it is quite improbable

to observe a cache miss at N_k^{int} when it is positioned at the root of the tree. One would observe a very high rate of traversals with low latency in this case, which decreases the expected traversal latency of N_k^{int} significantly. An accurate estimation for the cache misses requires the consideration of this particularity of the binary tree. To approximate the impact of this variation, we split N_k^{int} into a number (let H_k denotes this number for N_k^{int}) of independent virtual nodes (in the lines of independent reference model), each representing the behavior of N_k^{int} with a different *Read* rate. The virtual node, with *Read* rate $\lambda_{int,k,h}^{read}$, is denoted by $N_{h,int}^{k}$. We will obtain the *Read* rates $\lambda_{int,k,h}^{read}$ and presence probabilities $p_{k,h}^{int}$ for these virtual nodes by requiring that the average behaviors are still valid: $p_k^{int} = \sum_{h=1}^{H_k} p_{k,h}^{int}$ and $p_k^{int}\lambda_{int,k}^{read} = \sum_{h=1}^{H_k} p_{k,h}^{int}\lambda_{int,k,h}^{read}$.

Theorem 1. *For an external binary tree with N internal nodes, generated with the random permutation of insertions, the probability mass function of the size of the subtree (the random variable concerns only the number of the internal nodes and denoted by Sub_k) that is rooted at N_k^{int} is given by: $\mathbb{P}[Sub_k = N] = 1/N$ and $\mathbb{P}[Sub_k = s] = O(1/s^2)$.*

Proof. It is clear that $\mathbb{P}[Sub_k = N] = 1/N$ since it occurs iff N_k^{int} has the highest priority among all internal nodes. For the rest, we consider four different cases. Let σ_k denotes the index of N_k^{int} in the permutation of the sequence of N internal nodes that are arranged in the ascending order based on their keys.

(i) $\sigma_k + s \leq N$ and $\sigma_k - s \geq 1$: then there exist s distinct pairs of (N_j^{int}, N_i^{int}) such that $\sigma_i - \sigma_j = s + 1$ and $\sigma_j < \sigma_k < \sigma_i$. Given a pair of such (N_j^{int}, N_i^{int}), $Sub_k = s$ if the priorities of N_j^{int} and N_i^{int} are higher than the priorities of all N_x^{int}, such that $\sigma_j < \sigma_x < \sigma_i$ and also N_k^{int} has a higher priority than all $N_{y\neq k}^{int}$ such that $\sigma_j < \sigma_y < \sigma_i$. This ($N_k^{int}$ is the root of subtree that includes all N_y^{int}, such that $\sigma_j < \sigma_y < \sigma_i$) can happen with probability, $\frac{2}{(s+2)(s+1)s}$. There exist s such non-overlapping cases. We have, $\mathbb{P}[Sub_k = s] = \frac{2}{(s+1)(s+2)}$.

(ii) $\sigma_k + s > N$ and $\sigma_k - s \geq 1$: then there exist a N_i^{int} such that $\sigma_i = N - s$. $Sub_k = s$ if N_i^{int} has higher priority than all N_x^{int}, such that $\sigma_i < \sigma_x \leq N$ and N_k^{int} has higher priority than all N_y^{int}, such that $\sigma_i < \sigma_{y\neq k} \leq N$. This can

happen with probability, $\frac{1}{(s+1)s}$. In addition, there can be at least 0 and at most $s-1$ distinct pairs of (N_j^{int}, N_i^{int}) such that $\sigma_i - \sigma_j = s+1$ and $\sigma_j < \sigma_k < \sigma_i$. We have: $\frac{1}{(s+1)s} \leq \mathbb{P}[Sub_k = s] \leq \frac{1}{(s+1)s} + \frac{2(s-1)}{(s+1)(s+2)s}$.

(iii) $\sigma_k + s \leq N$ and $\sigma_k - s < 1$: The bound at (ii) applies to this case also.

(iv) $\sigma_k + s > N$ and $\sigma_k - s < 1$: then there exist a N_i^{int} such that $\sigma_i = N - s$ and a N_j^{int} such that $\sigma_j = s+1$. In addition, there can be at least 0 and at most $s-2$ distinct pairs of nodes (N_j^{int}, N_i^{int}) such that $\sigma_i - \sigma_j = s+1$ and $\sigma_j < \sigma_k < \sigma_i$. Similar to (i) and (ii), we obtain and sum the probabilities lead to $Sub_k = s$. We have: $\frac{2}{(s+1)s} \leq \mathbb{P}[Sub_k = s] \leq \frac{2}{(s+1)s} + \frac{2(s-2)}{(s+1)(s+2)s}$ □

We start with an observation. The *Read* rate of N_k^{int} is proportional to the size of the subtree that is rooted at N_k^{int}. Given a binary tree of N internal nodes, the size of the subtree can vary in the interval $[1, N]$, which means that we can have $H_k = N$ different *Read* rate levels ($\lambda_{int,k,h}^{read}$) associated with their presence probabilities $p_{k,h}^{int} = p_k^{int}\mathbb{P}[Sub_k = h]$. Relying on Theorem 1, one can observe that $\mathbb{P}[Sub_k = h]$ do not variate much from $c_1/(h+1)^2$ for the majority of different values of h and k. Therefore, we approximate $\mathbb{P}[Sub_k = h] \approx c_1/(h+1)^2$, with a single constant c_1 for all k and $h < H_k$. We know, $\sum_{h=1}^{H_k} \mathbb{P}[Sub_k = h] = 1$ and $\mathbb{P}[Sub_k = H_k] = 1/H_k$. So, we obtain $c_1 \approx 2$ by solving the equation $\int_{h=2}^{N}(c_1/h^2)dh = (N-1)/N$. We set $p_{k,h}^{int} = p_k^{int}(2/(h+1)^2)$ and $p_{k,H_k}^{int} = p_k^{int}/H_k$. Assuming $\lambda_{int,k,h}^{read} = c_2 h \lambda_{int,k}^{read}$ (*Read* rates are proportional to the subtree size), we require $p_k^{int}\lambda_{int,k}^{read} = \sum_{h=1}^{H_k} p_{k,h}^{int}\lambda_{int,k,h}^{read}$, which leads to $\lambda_{int,k}^{read} \approx c_2 + \int_{h=2}^{H_k}(2/h^2)c_2(h-1)\lambda_{int,k}^{read}dh$. We solve and obtain $c_2 \approx 1/(2\ln H_k)$. We set $\lambda_{int,k,h}^{read} = h\lambda_{int,k}^{read}/(2\ln H_k)$, for the virtual internal nodes.

Now, we consider the *CAS* events. Delete and Insert operation start with the search phase. Insert operation finalize with a *CAS* executed at the grandparent internal node of the inserted external key. Delete operation contains three *CAS*; (i) one at the grandparent internal node of the deleted external key; (ii) two that are executed consecutively at the parent node of the external key. Thus, we consider them as a single *CAS* instruction, since the second of the consecutive ones has a negligible cost because the cacheline has already been exclusively owned by the thread.

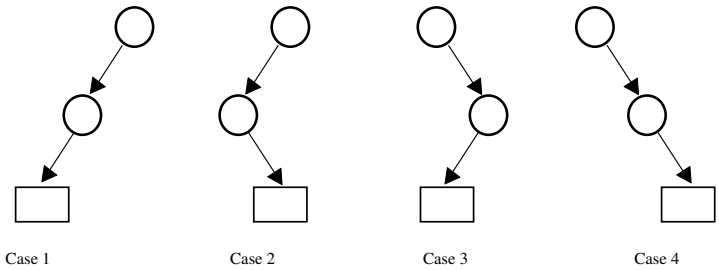


Figure 5.6: Binary Tree CAS Probability

Similar to *Read* events, we first find the rate of *CAS* events for N_k^{int} and split these events to virtual nodes by requiring the average behavior is still valid: $p_k^{int}\lambda_{int,k}^{cas} = \sum_{h=1}^{H_k} p_{k,h}^{int}\lambda_{int,k,h}^{cas}$. To determine the target of *CAS* event, we need to determine the probability of an internal node N_k^{int} to be the grandparent or parent of the targetted $N_{k'}^{ext}$. We examine four different cases as illustrated in Figure 5.6. Given that we are in the first case, we look for the probability that $N_k^{int}, k' < k$, to possess the smallest or second smallest key, that is bigger than k', among the internal nodes that are present in the tree. Such internal nodes with the smallest key and the second smallest key corresponds to the parent and grandparent of $N_{k'}^{ext}$, respectively. For case 1, it is possible that the grandparent node is the node which has the xth, $x > 1$, smallest key that is bigger than i, that is present in the tree. But this probability decreases exponentially as x increases. That is why, we have attributed the *CAS* events that takes place at the granparent node to the node with second smallest key that is bigger than k'. For case 2, the parent corresponds to the smallest key that is bigger than k' and the grandparent corresponds to the biggest key that is smaller than k', that are present in the tree.

Formally, let $P_{k'}^B = \{i : i \geq k', N_i^{int} \in D\}$ and $P_{k'}^S = \{i : i < k', N_i^{int} \in D\}$. For the first case, we are interested in the probability that N_k^{int} is the grandparent or parent node of $N_{k'}^{ext}$. These are given by $\mathbb{P}\left[k = \sup\{P_{k'}^S - \sup\{P_{k'}^S\}\}\right]$ and $\mathbb{P}\left[k = \sup\{P_{k'}^S\}\right]$ respectively. For the second case, we are interested in $\mathbb{P}\left[k = \sup\{P_{k'}^S\}\right]$ and $\mathbb{P}\left[k = \inf\{P_{k'}^B\}\right]$. The third and fourth cases follows

the same lines as they are the flipped versions of the case one and two. For all non-sentinel nodes, we have $p_k^{int} = p$. First, we compute the following probabilities:

For $k \geq k'$ we have: (these probabilities are zero if $k < k'$)

$$\mathbb{P}\left[k = \sup\{P_{k'}^S - \sup\{P_{k'}^S\}\}\right] = p(k' - i)(1-p)^{(k'-k-1)}$$

$$\mathbb{P}\left[k = \sup\{P_{k'}^S\}\right] = (1-p)^{(k'-k)}$$

And for $k < k'$: (these probabilities are zero if $k \geq k'$)

$$\mathbb{P}\left[k = \inf\{P_{k'}^B\}\right] = (1-p)^{(k-k'-1)}$$

$$\mathbb{P}\left[k = \inf\{P_{k'}^B - \inf\{P_{k'}^B\}\}\right] = p(k - k' - 1)(1-p)^{(k-k'-2)}$$

Based on Lemma 18 (assuming a constant tree size), we obtain the expected number of internal nodes that route the search to its left child ($c_{k',l}$) and right child($c_{k',r}$) for an operation that is targetted to $key = k'$. On this route, we compute the probability of a random node to be the left (right) child of its parent, with $l_{k'} = c_{k',l}/(c_{k',l} + c_{k',r})$ (and similarly $r_{=}c_{k',r}/(c_{k',l} + c_{k',r})$). And, we estimate the probability of observing a case at a random time by using these values (*i.e.* $l_{k'}^2$ for Case 1, $l_{k'}r_{k'}$ for Case 2). And finally, we obtain:

$$\begin{aligned}\mathbb{P}\left[op_{k'}^{del} \rightsquigarrow cas\big(N_k^{int}\big)\right] =& p_{k'}^{int}(l_{k'}^2\mathbb{P}\left[k = \inf\{P_{k'}^B - \inf\{P_{k'}^B\}\}\right] \\ &+ l_{k'}(r_{k'}+1)\mathbb{P}\left[k = \inf\{P_{k'}^B\}\right] \\ &+ r_{k'}(l_{k'}+1)\mathbb{P}\left[k = \sup\{P_{k'}^S\}\right] \\ &+ r_{k'}^2\mathbb{P}\left[k = \sup\{P_{k'}^S - \sup\{P_{k'}^S\}\}\right])\end{aligned}$$

$$\begin{aligned}\mathbb{P}\left[op_{k'}^{ins} \rightsquigarrow cas\big(N_k^{int}\big)\right] =& (1 - p_{k'}^{int})(l_{k'}^2\mathbb{P}\left[k = \inf\{P_{k'}^B - \inf\{P_{k'}^B\}\}\right] \\ &+ l_{k'}r_{k'}\mathbb{P}\left[k = \inf\{P_{k'}^B\}\right] \\ &+ r_{k'}l_{k'}\mathbb{P}\left[k = \sup\{P_{k'}^S\}\right] \\ &+ r_{k'}^2\mathbb{P}\left[k = \sup\{P_{k'}^S - \sup\{P_{k'}^S\}\}\right])\end{aligned}$$

Lastly, we split the *CAS* events to the virtual nodes. *CAS* events can happen at the internal nodes only when they are in the last two levels of the tree (or similarly when the size of the subtree that is rooted at the concerned internal node is in the interval $[1, 3]$). We required the average behaviour to be valid and set $\lambda^{cas}_{int,k,x} = p^{int}_{k} \lambda^{cas}_{int,k} / (p^{int}_{k,1} + p^{int}_{k,2} + p^{int}_{k,3}), \forall x \in \{1, 2, 3\}$. For the cases where the operation key selection follows a zipf distribution, there exist a small region of the tree that the most operations concentrate. The update operations concentrate to that region so that the nodes are expected to change levels frequently. This means that the impact of invalidation recovery factor can be seen while the node is at an level. For this impacting factor, for zipf distribution, we split the events to virtual nodes evenly, $\forall h, \lambda^{cas}_{int,k,h} = \lambda^{cas}_{int,k}$.

5.7 Experimental Evaluation

We validate our model through a set of well-known lock-free search data structure designs, mentioned in the previous section. We stress the model with various access patterns and number of threads to cover various scenarios where the data structures could be exploited. For the key selection process, we vary the key ranges and the distribution: from uniform (*i.e.* the probability of targeting any key is constant for each operation) to zipf (with $\alpha = 1.1$ and the probability to target a key decreases with the value of the key). Regarding the operation types, we start with various balanced update ratios, *i.e.* such that the ratio of Insert (among all operations: Search, Delete, Insert) is equal to the ratio of Delete. Then, we also consider asymmetric cases where the ratio of Insert and Delete operations are not equal, which changes the expected size of the structure.

5.7.1 Setting

We have conducted experiments on an Intel ccNUMA workstation system. The system is composed of two sockets, each containing eight physical cores. The system is equipped with Intel Xeon E5-2687W v2 CPUs. Threads are pinned

to separate cores. One can observe the performance change when number of threads exceeds 8, which activates the second socket.

In all the figures, y-axis provides the throughput, while the number of threads is represented on x-axis. The dots provide the results of the experiments and the lines provide the estimates of our framework. The key range of the data structure is given at the top of the figures and the percentage of update operations are color coded.

We instantiate all the algorithm and architecture related latencies, following the methodologies described in [23, 24]. In line with these studies, we observed that the latencies of t^{cas} and t^{rec} are based on thread placement. We distinguish two different costs for t^{cas} according to the number of active sockets. Similarly, given a thread accessing to a node N_i, the recovery latency is low (resp. high), denoted by t^{rec}_{low} (resp. t^{rec}_{high}), if the modification has been performed by a thread that is pinned to the same (resp. another) socket. Before the execution, we measure both t^{rec}_{low} and t^{rec}_{high}, and instantiate t^{rec} with the average recovery latency, computed in the following way for a two-socket chip. For $s \in \{1, 2\}$, we denote by P_s the number of threads that are pinned to socket numbered s. By taking into account all combinations, we have $t^{rec} = (P_1(P_1 t^{rec}_{low} + P_2 t^{rec}_{high}) + P_2(P_2 t^{rec}_{low} + P_1 t^{rec}_{high}))/P^2$. Since $P = P_1 + P_2$, we obtain $t^{rec} = t^{rec}_{low} + 2(P_1/P)(1 - P_1/P)(t^{rec}_{high} - t^{rec}_{low})$.

For the data structure implementation, we have used ASCYLIB library [12] that is coupled with an epoch based memory management mechanism which introduces negligible latency.

5.7.2 Search Data Structures

5.7.2.1 Linked List

Figures 5.7, 5.8 and 5.9 illustrates the results for the lock-free linked list, for various scenarios that are described before (see 5.7).

Independent Reference Model assumes that the occurrence patterns of events at the different nodes follow independent Poisson processes. However, the sequence of node accesses in an linked list operation reveals a high degree of

dependence, implying that the Poisson processes for the different nodes are indeed dependent. Also, we hypothesize the set of traversed nodes in an operation is small in front of the set of all nodes, which leads to the rareness of events, that makes the Poisson process approximation for Bernoulli processes well-conditioned. All these imply that linked list might not be expressed in the model. However, we still involve linked list in our study to see what happens when we diverge from our modeling assumptions.

In figures, we observe that our approach indeed fail (underestimate) to capture the capacity cache misses. This is revealed clearly in the cases in which the expected size of the linked list is around the cache sizes. It is not apparent for some cases since the underestimation does not impact the outcome if the cache miss ratio is already low. On the other hand, *Compare-and-Swap* related impacting factors are estimated accurately.

For a given node, the capacity cache miss estimation requires a collective approach that involves all the nodes. In contrast, *Compare-and-Swap* related impacting factors (coherence misses) are estimated based on the events on the concerned node. The sequence of events in a node follow approximately a Poisson process (See Section 5.4.2) even the processes in different nodes are dependent. This is the reason why our approach manages to provide better estimations in the cases where the modification related factors dominate the performance. This can be observed when the curves with different update rates (colors) diverge from each other, implying the significance of the modifications on the performance.

We believe these partially negative results create a reference point to evaluate the accuracy of estimations for the other data structures.

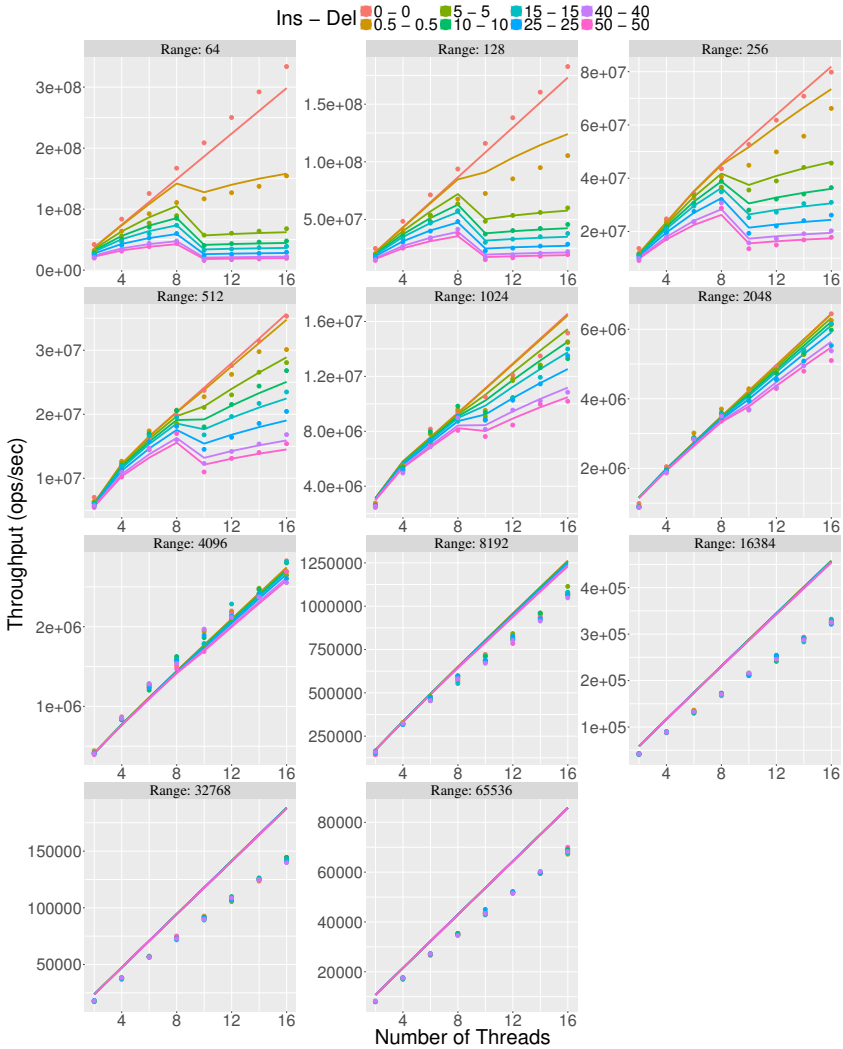


Figure 5.7: LL Uniform distribution for key selection

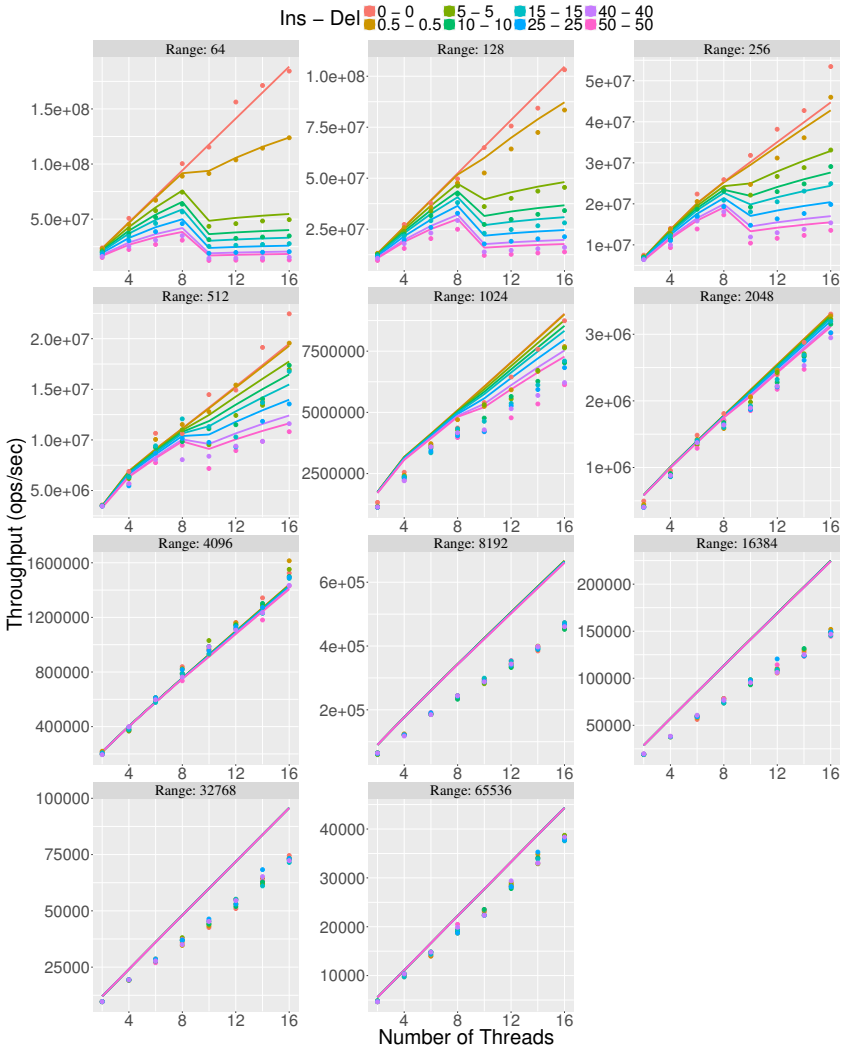


Figure 5.8: LL Zipf distribution for key selection

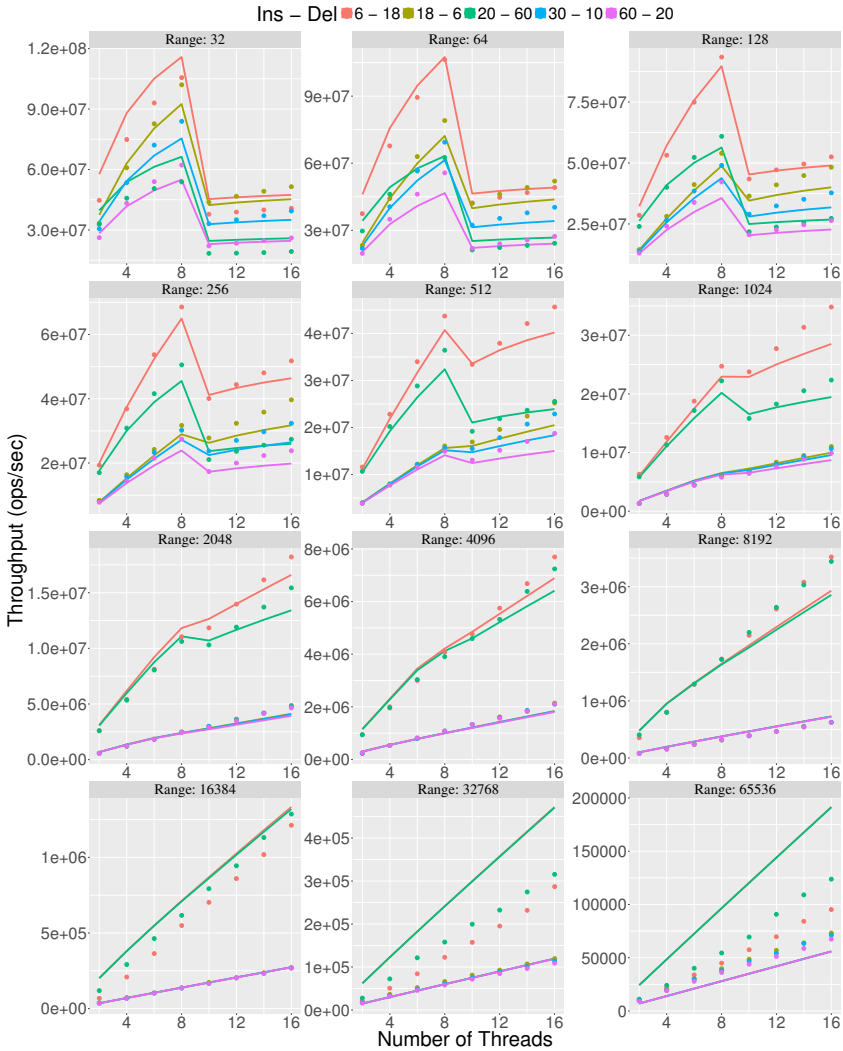


Figure 5.9: LL asymmetric update rates, uniform distribution for key selection

5.7.2.2 Hash Table

Figure 5.10, 5.11 and 5.12 illustrates the results for the lock-free hash table with different load factor values (number of slots per bucket) where the key selection process is initiated with uniform distribution. Figure 5.13 shows the results of a case where the selection process follows the Zipf distribution. Lastly, Figure 5.14 reveals the results for asymmetric delete and insert operation ratios where the key selection is done with the uniform distribution.

One can see that the performance drops as the update rate increases, due to the impact of *CAS* related factors. This impact magnifies with the activation of the second socket (more than eight threads) since the events become more costly. When there is no update operation, the performance scales linearly with the number of threads. This is also observed when the percentage of updates are low. However, we lose this scalability especially in the regions where the number of threads is more than eight because of the impact of coherence misses, *e.g.* a thread encounters invalidated nodes (with a high recovery latency) more frequently in its traversal. Also, performance decreases as the key range of hash table increases because of capacity cache misses.

These effects are captured accurately by our framework, and our estimates follow the real behavior almost for all cases that we consider.

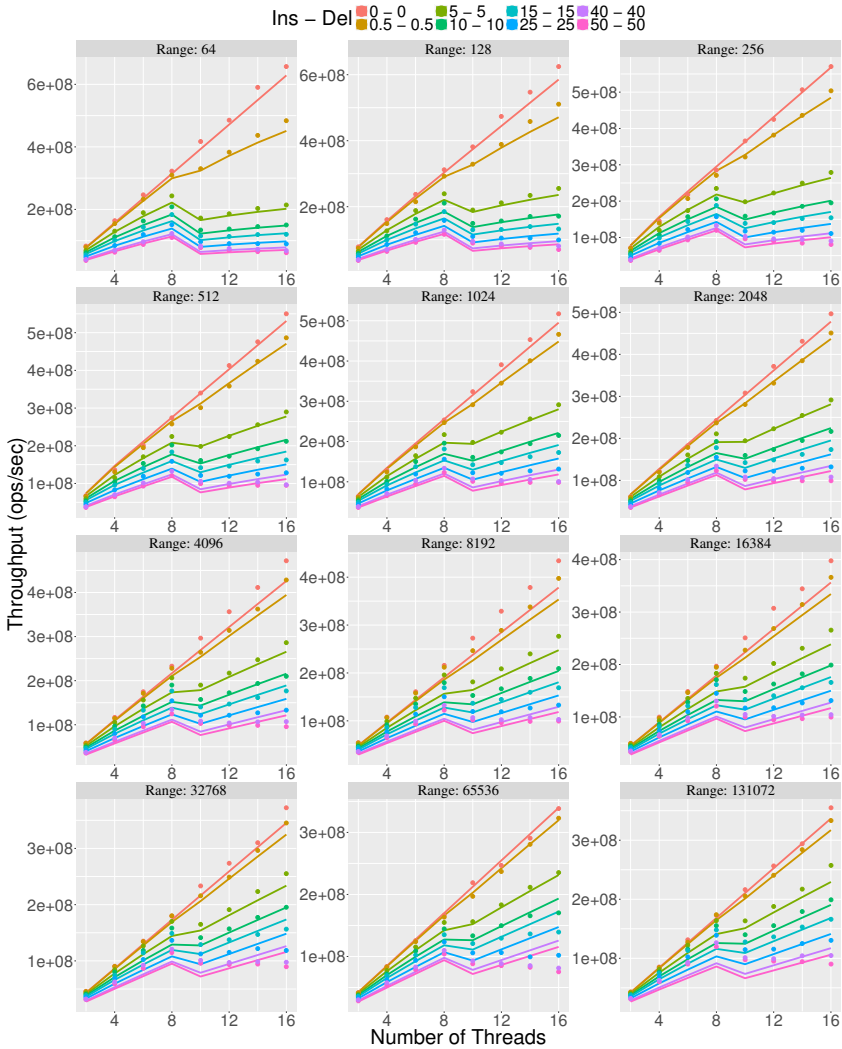


Figure 5.10: HT Uniform distribution for key selection, with load factor=2

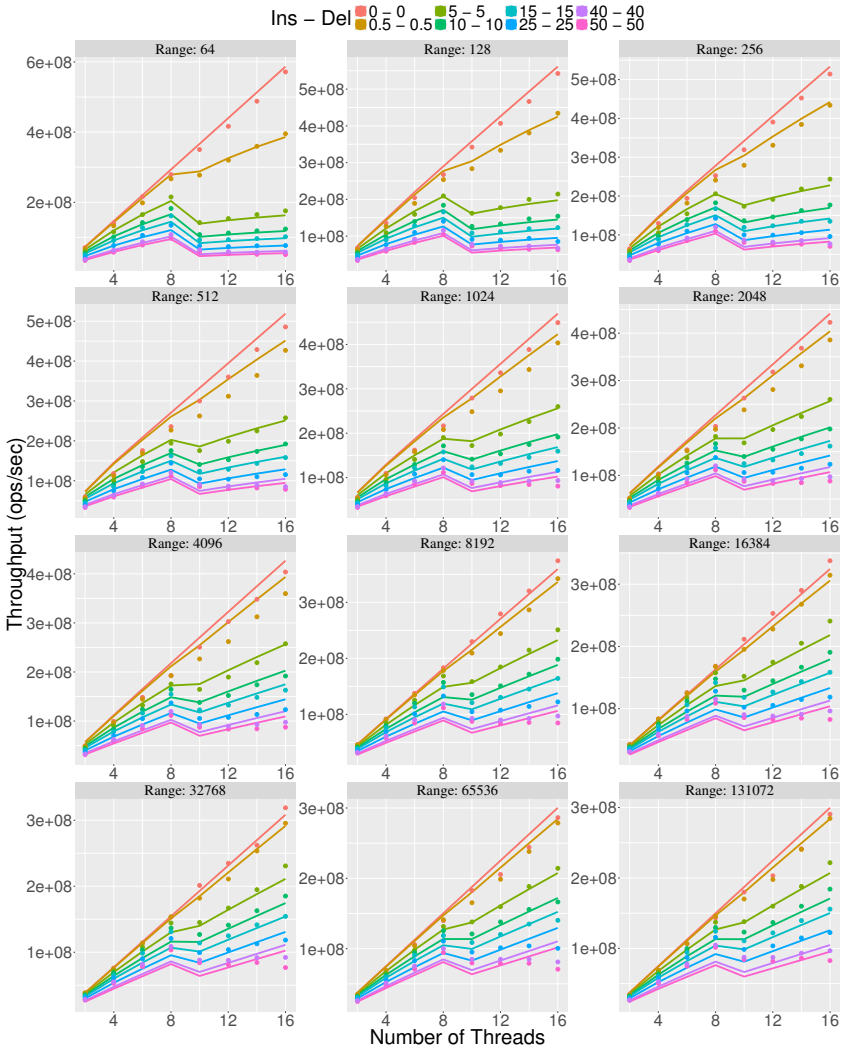


Figure 5.11: HT Uniform distribution for key selection, with load factor=4

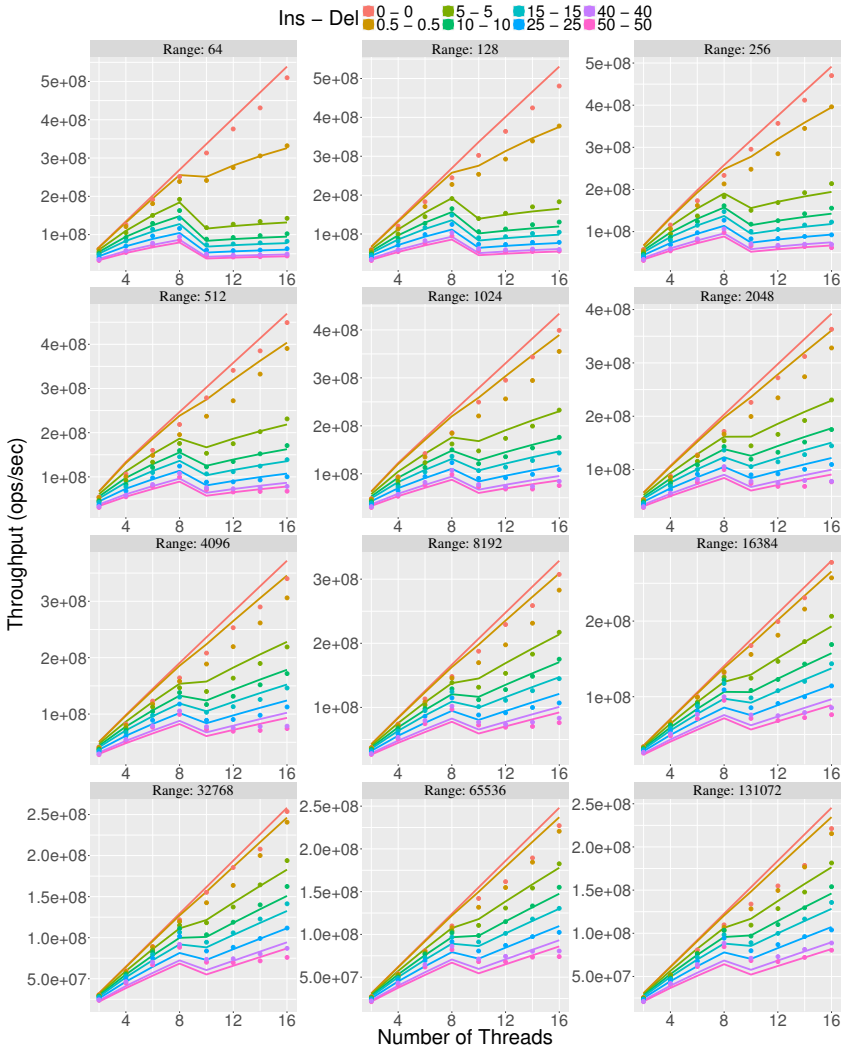


Figure 5.12: HT Uniform distribution for key selection, with load factor=8

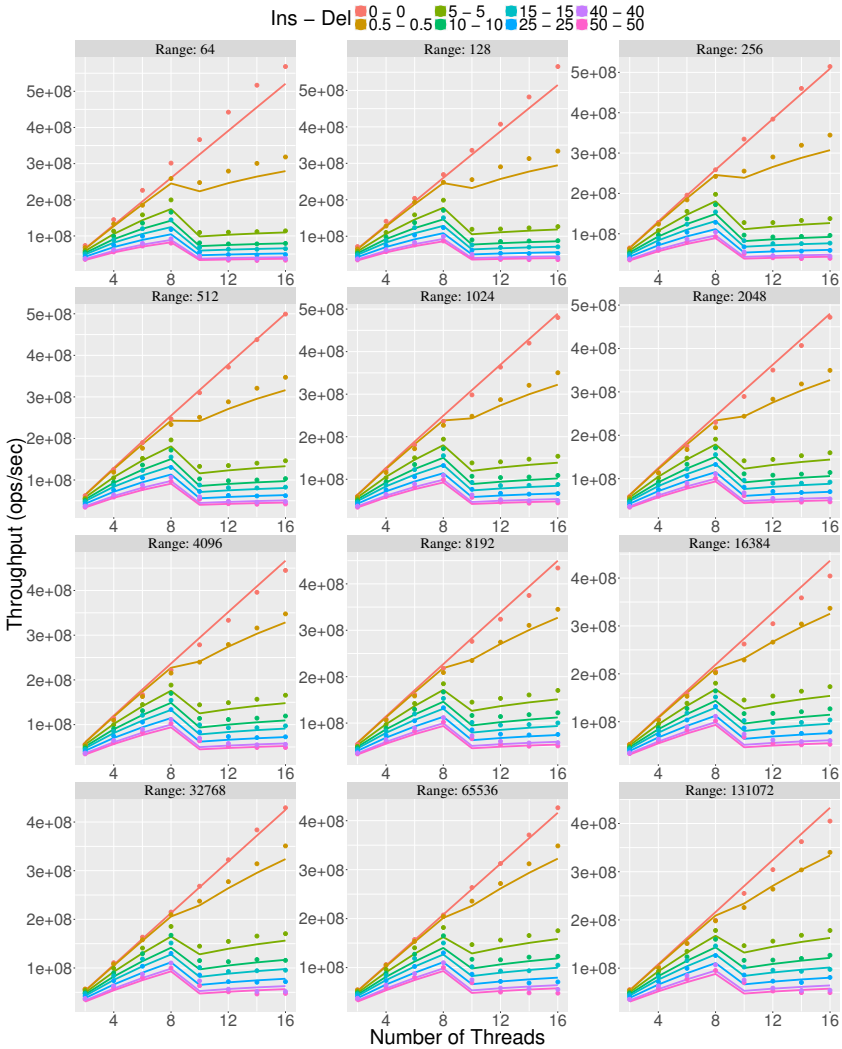


Figure 5.13: HT Zipf distribution for key selection, with load factor=2

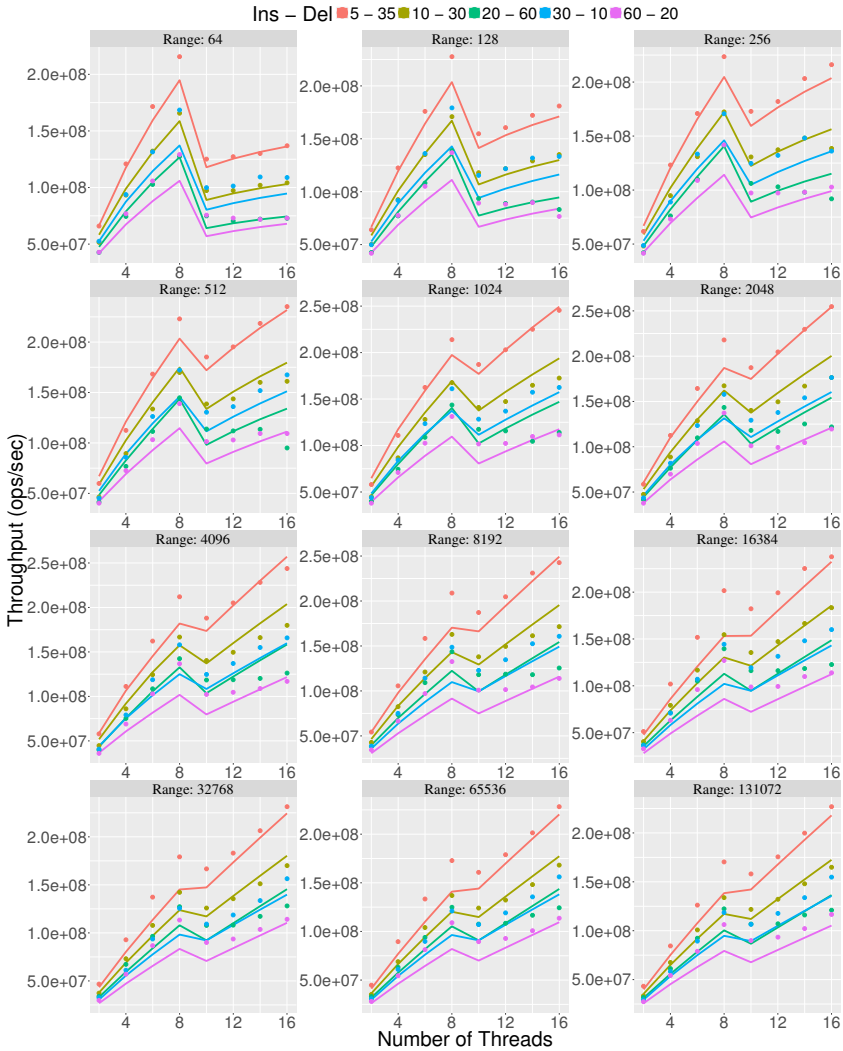


Figure 5.14: HT asymmetric update operations, Uniform distribution for key selection, with load factor=4

5.7.2.3 Skip List

Figure 5.15, 5.16 and 5.17 illustrates the results for the lock-free skip list, for various scenarios that are described before (see 5.7), where the estimations often closely follow the real behavior. In Figure 5.17, we observe that our estimation show some deviation from the real behavior when the key range is small, and Delete ratio is higher than Insert. For such cases, the expected size of search data structure tends to be very small which might lead to inaccuracies. Also, when the update rate is very high and the key selection is made with the Zipf distribution, we overestimate the performance, presumably because we ignore the retry loop conflicts that might appear in such extreme cases to some extent.

The events on some nodes (such as the nodes with the maximum height or also applies to the root of a tree) are not in the lines of our model (rareness and independent reference). However, this does not lead to inaccuracies in performance estimations. These nodes reveal very high read rates; therefore they almost always occupy the lowest level cache, and any modification on them is observed almost instantly by other threads before any other modification takes place. Even the interarrival times of events in such nodes are not approximately exponentially distributed, the traversal latency of them is estimated accurately. Once the average traversal frequency is also estimated accurately too (we do), such nodes do not lead to inaccuracies. Also, this does not influence the capacity miss ratio estimation of other nodes since such nodes are few and, as mentioned before, they have a very high read rate which makes their cache residence predictable by any approach.

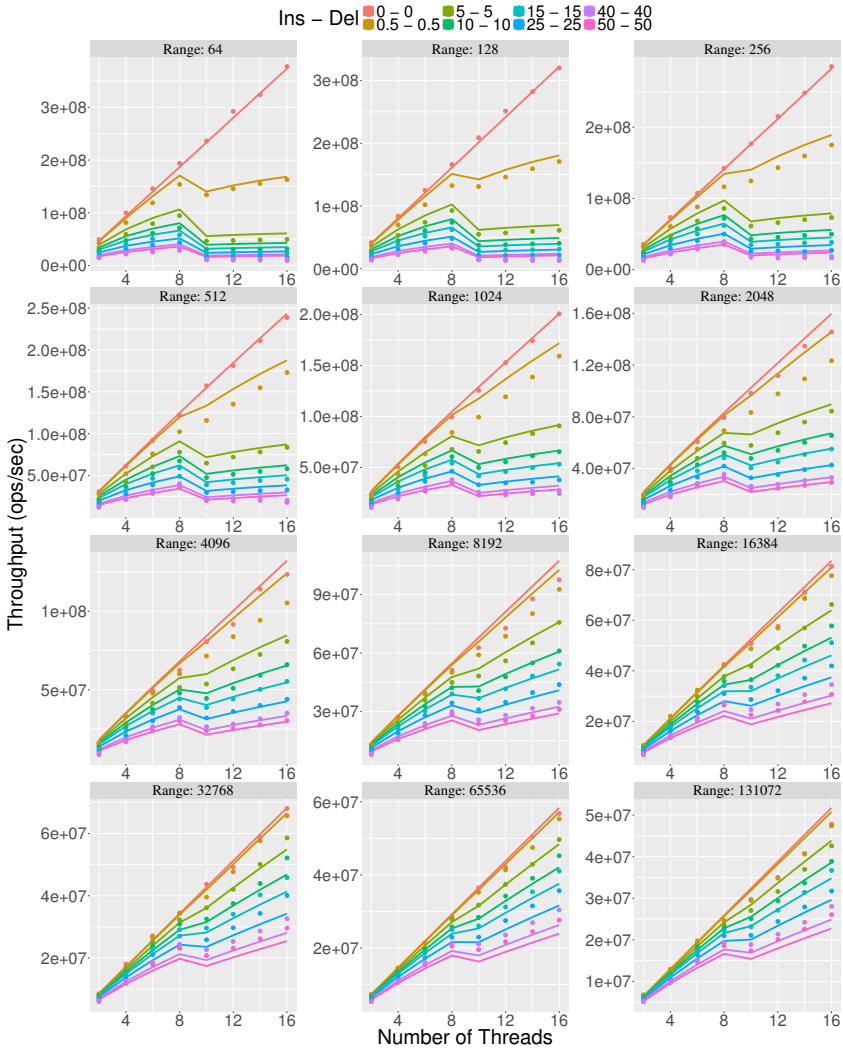


Figure 5.15: Skiplist Uniform distribution for key selection

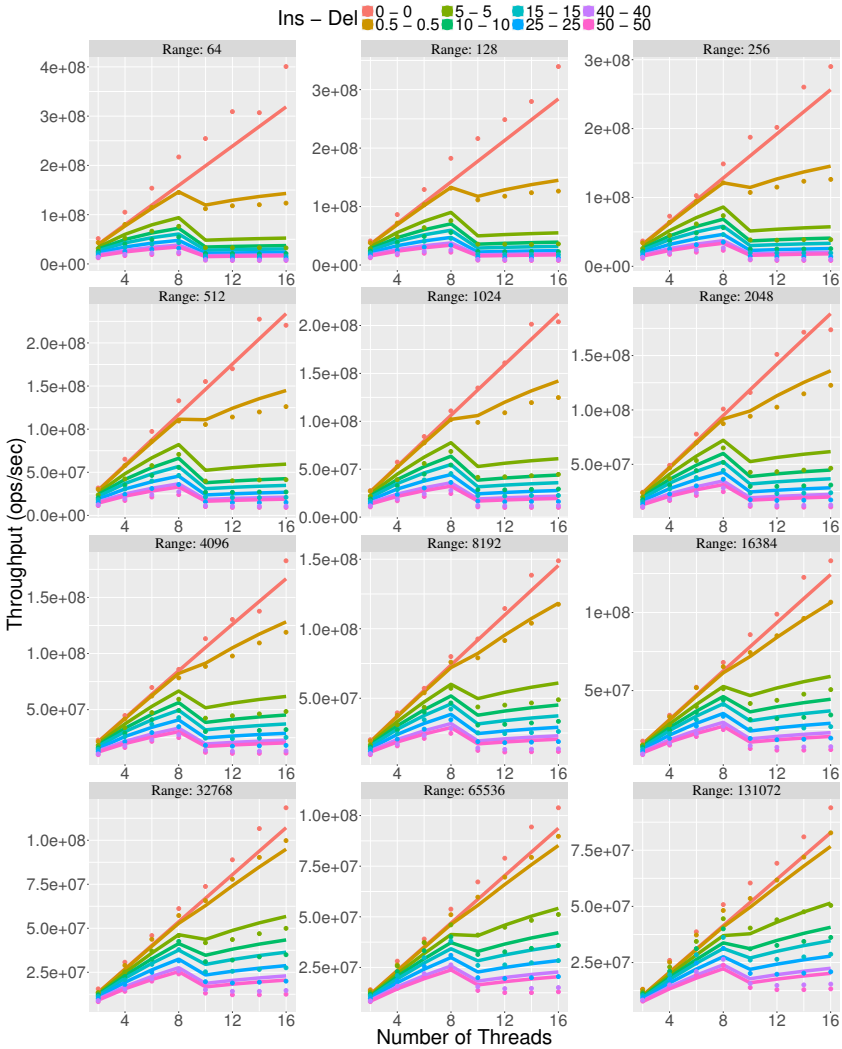


Figure 5.16: Skiplist Zipf distribution for key selection

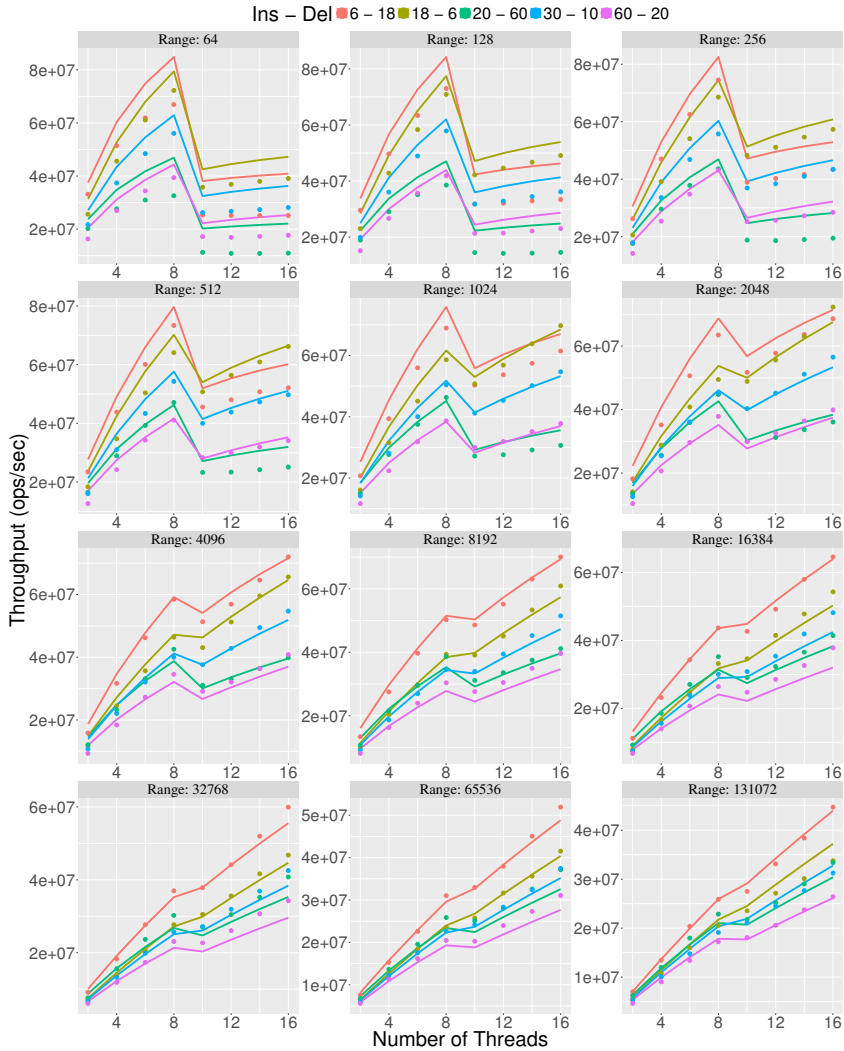


Figure 5.17: Skiplist asymmetric update rates, uniform distribution for key selection

5.7.2.4 Binary Tree

Figure 5.18, 5.19 and 5.20 illustrates the results for the binary tree, for various scenarios that are described before (see 5.7). Here, we observe that our estimations often closely follow the real behaviour.

Discussing the difference between binary tree and skip list would be interesting. Loosely speaking, these two data structure provide the same asymptotic complexity in average, but we can observe in practice that binary tree outperforms skip list in the majority of configurations.

One can observe that binary tree outperforms skip list when there are no updates. We conjecture that skip list operation traverses more nodes in average. It searches in a level until it encounters a key that is bigger (or equal if lucky) than the search key. Then, it returns to the previous key to continue the traversal in the lower level. This additional node traversal (the one from which it comes back) at each level might be the leading actor of the observed performance difference.

It can be observed that binary tree scales better with the increase in the number of threads for the cases with high update rates. Firstly, skip list operations require more *Compare-and-Swap* executions on average, therefore performs slower than binary tree. Secondly, binary tree modifications occur only in the leaves, where the read rates of nodes are low. Thus, it is less probable to observe a modification before another overwrites it. This effect can be observed when one compares the experiments with the Zipf and the uniform distribution for binary tree. The scalability (with respect to increasing number of threads) is lost with Zipf because threads operate on the same portion of the tree, which more makes it more probable for a thread to observe a modification before it is overwritten.

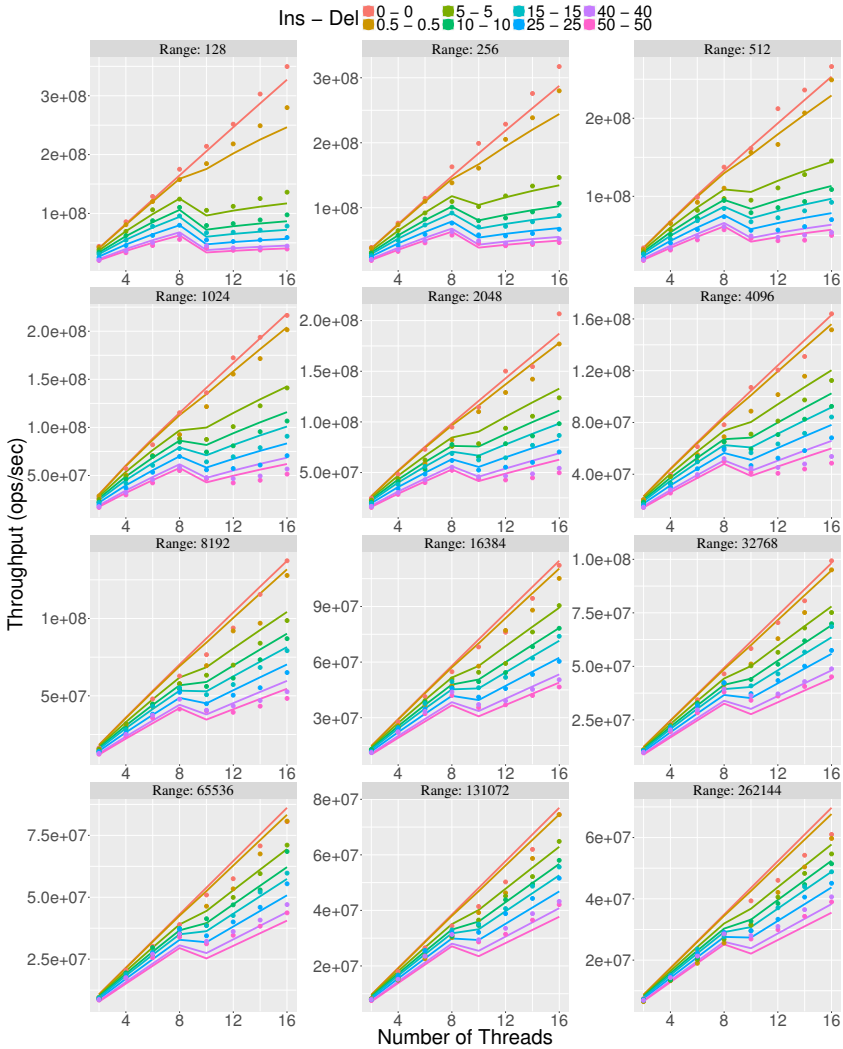


Figure 5.18: BST Uniform distribution for key selection

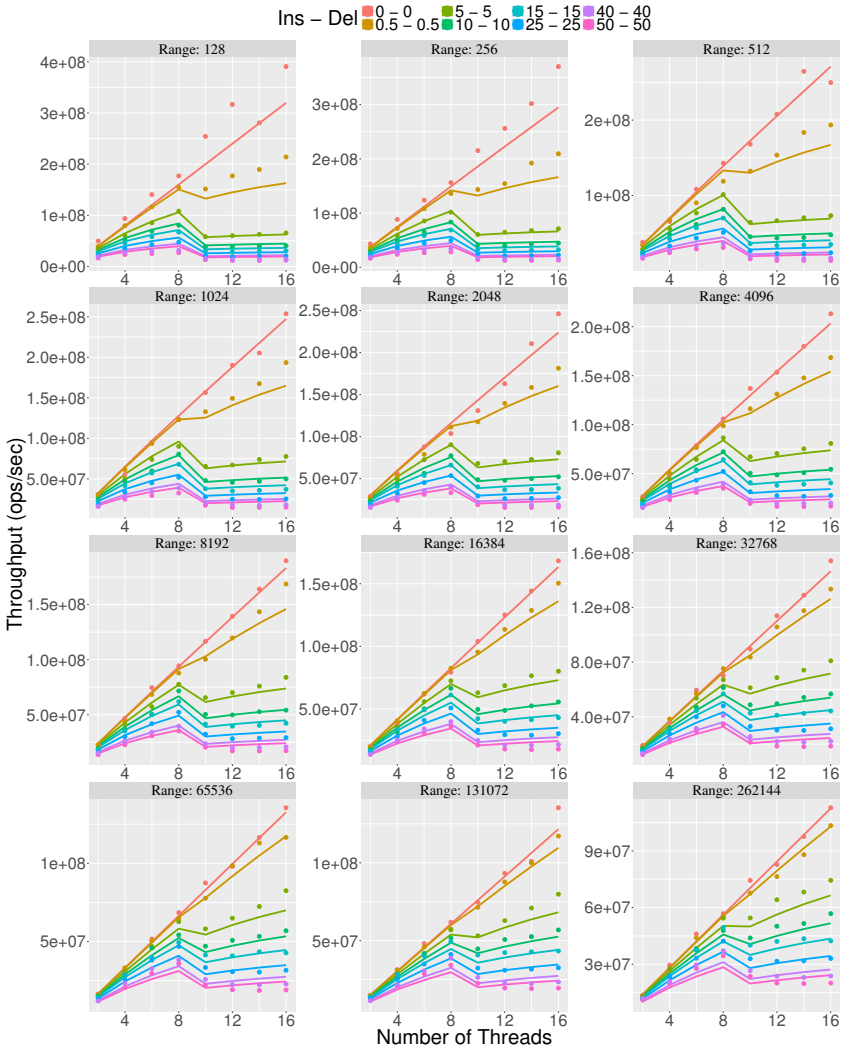


Figure 5.19: BST Zipf distribution for key selection

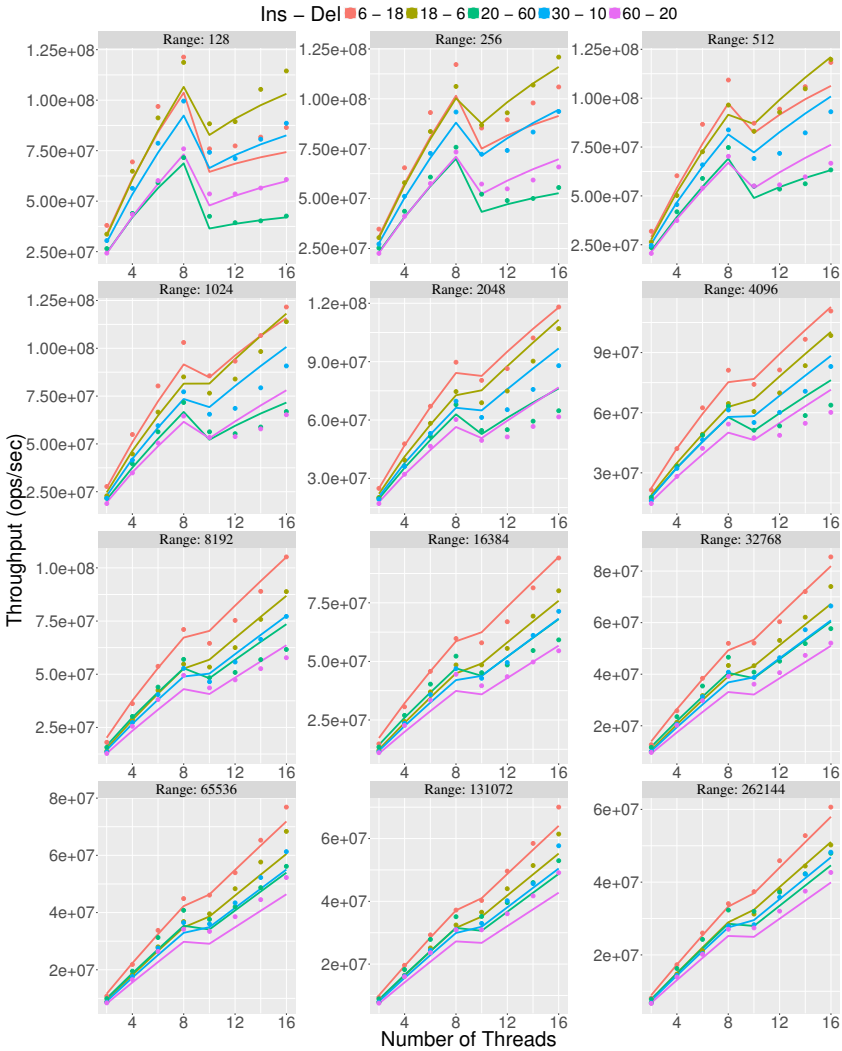


Figure 5.20: BST asymmetric update rates, uniform distribution for key selection

5.8 Applications: to Pad or not to Pad

In a non-padded (packed) configuration, multiple nodes are packed together into a single cacheline. This implies that a modification done at a node, could lead to a coherence cache miss in the traversal of the other nodes. It is often referred as *false sharing*. On the other hand, the packed configurations benefit from their compact representation by reducing the capacity misses.

Until now, we have assumed that the nodes are padded. Here, we extend the framework to estimate the performance of a packed configuration to facilitate the tuning process. In such a setting, where the nodes are inserted and deleted repeatedly, N_i can be alone in its cacheline with the old versions of a set of nodes that are not present any more in the data structure. Alternatively, it might be mapped to the same cacheline with some number of active nodes that are present in the search data structure and they all together contribute to the event rates that are originating from the same cacheline.

Firstly, we assume that at most two nodes can be packed to a cacheline (that is the case for the data structures that we consider). We denote the total number of slots for the node allocations with $S = 2\mathcal{M}pageSize/cacheLineSize$. Recall that $\mathcal{M}$ is the number of pages that are used by the structure. We assume that the nodes are assigned uniformly to the slots; given that N_i and N_j are present in the structure, N_j is mapped to the same cacheline as N_i with probability: $1/(S-1)$. With the linearity of expectation, the expected additional event rate for the cacheline that N_i is mapped to can be given by the sum of event rates originating from different nodes. λ_i^{read} and λ_i^{cas} provides the event rates for N_i, and we introduce an additive factor to represent the average event rate contributions of other nodes to the cacheline of N_i: $\lambda_i^{read,addi}$ for *Read* events, and $\lambda_i^{cas,addi}$ for *CAS* events. N_j contribute to the *Read* event rates with λ_j^{read} if N_j and N_i are assigned to the same cacheline, which happens with probability $p_j/(S-1)$. Then, we have: $\lambda_i^{read,addi} = \sum_{j=1,j\neq i}^{\mathcal{N}} \lambda_j^{read}(\frac{p_j}{S-1})$ and $\lambda_i^{cas,addi} = \sum_{j=1,j\neq i}^{\mathcal{N}} \lambda_j^{cas}(\frac{p_j}{S-1})$.

With the node packing, we obtain additive components for *CAS* and *Read* events. Now, we show the integration of these additive components into the

process.

5.8.0.1 Cache Misses

To begin with, packing would have a positive impact on the cache misses as it would increase the characteristic time (T) of the cache, that is the duration for C unique cacheline references. To recall, N_i could contribute to this C references only if $N_i \in D$ and we have embedded this effect into the process by introducing the random variable P_i (see 5.5.1.5). With the packing, this contribution becomes less probable, as the contribution would occur only if the reference to N_i occurs before the references to the other node that is mapped to the same cacheline with N_i. Otherwise, the reference to N_i would be ineffective for the characteristic time. To recall, the characteristic time is the solution of the following equation:

$$X^{cache}(t) = \sum_{j=1}^{N} P_i^{pack} \mathbf{1}_{0<O_j \leq t}$$

where P_i^{pack} is the variable that we modify in the process,

$$P_i^{pack} = \begin{cases} p_i(\lambda_i^{read}/(\lambda_i^{read} + \lambda_i^{read,addi})), & \text{if } P_i^{pack} = 1 \\ 1 - p_i(\lambda_i^{read}/(\lambda_i^{read} + \lambda_i^{read,addi})), & \text{if } P_i^{pack} = 0 \end{cases}$$

Having obtained the characteristic time, we involve the additive factor to estimate the cache miss rate of N_i. This is because a reference leads to a cache miss (in a cache of size C) only if the previous C cacheline references do not include the cacheline that N_i is mapped to.

$$Hit_i^{cache} = 1 - e^{(-(\lambda_i^{read} + \lambda_i^{read,addi}))/P)T}$$

5.8.0.2 Page Misses

Secondly, packing can improve the TLB cache hit ratios. This simply happens because it reduces the total number of pages that the search data structure spans.

To recall, the total number of pages is a parameter of the process that computes the expected latency for the impacting factor (Hit_i^{tlb}). Packing do not influence the process, so we just need to update the value of the parameter.

5.8.0.3 CAS Execution

On the downside, packing is expected to reduce the performance through the *CAS* related impacting factors. To recall, CAS_i^{reco} represents the expected latency per traversal at N_i for executing *CAS* instructions targeted to N_i. This factor is proportional to the throughput, and packing do not change the probability of executing a *CAS* at N_i while traversing it. So, packing does not have a direct impact on this component.

5.8.0.4 Invalidation Recovery

The most important performance impacting *CAS* related factor is the invalidation recovery. For each traversal of N_i, there exist a possibility to pay for a coherence cache miss due to the previous *CAS* executions at the cacheline, that N_i is mapped to. To compute the probability of a coherence miss, one needs to consider the previous events on the cacheline. The traversal (by a thread at N_i) would not experience the coherence miss if the previous traversal (on the cacheline that N_i is mapped to) of the same thread is not followed by *CAS* event of another thread. Thus, we consider the additive factor for both type of events and modify the process as follows:

$$= \mathbb{P}\left[\text{Coherence Miss on traversal of } N_i\right]$$
$$\frac{(\lambda_i^{cas} + \lambda_i^{cas,addi})(P-1)}{(\lambda_i^{cas} + \lambda_i^{cas,addi})P + (\lambda_i^{read} + \lambda_i^{read,addi})}$$

5.8.0.5 Stall Time

Finally, packing has a potential to increase the ratio of time that the cacheline (that N_i is mapped to) is blocked due to *CAS* executions. We simply update the process by involving the additive factor:

$$\mathbb{E}\left[CAS_i^{stall}\right] = (\lambda_i^{cas} + \lambda_i^{cas,addi})(P-1)t^{cas}\frac{t^{cas}}{2}$$

5.8.0.6 Experiments

In Figures 5.22 and 5.21, the results are depicted for configurations with padding (dashed lines), packing(dots) and our packing based estimations(lines), for the linked list and hash table (nodes for tree and skiplist is too large to be packed in a single cacheline or already packed). The key selection is done with the uniform distribution. For almost every case, we observe that the packing increases the performance and the performance do not degrade due to the false sharing, even when the update rate is high. The stall time ($\mathbb{E}\left[CAS_i^{stall}\right]$) often is not significant and the invalidation recovery ($\mathbb{E}\left[CAS_i^{reco}\right]$) dominates the performance when there are update operations. As an observation, the latency induced by this factor do not increase with packing, presumably because:

$$\frac{(\lambda_i^{cas} + \lambda_i^{cas,addi})(P-1)}{(\lambda_i^{cas} + \lambda_i^{cas,addi})P + (\lambda_i^{read} + \lambda_i^{read,addi})} \approx \frac{\lambda_i^{cas}(P-1)}{\lambda_i^{cas}P + \lambda_i^{read}}$$

This might explain us the reason why the false sharing do not degrade the performance, as opposed to one might expect. However, the cache and page misses influence the performance positively, as expected.

Our estimations show that these effects are captured by our framework. We observe a slight increase in almost all the curves that is coupled with a slight increase in our estimations, due to the reduced capacity cache misses.

5.9 Conclusion

In this paper, we have modeled and analyzed the performance of search data structures under a stationary and memoryless access pattern. We have distinguished two types of events that occur in the search data structure nodes and have modeled the arrival of events with Poisson processes. The properties of

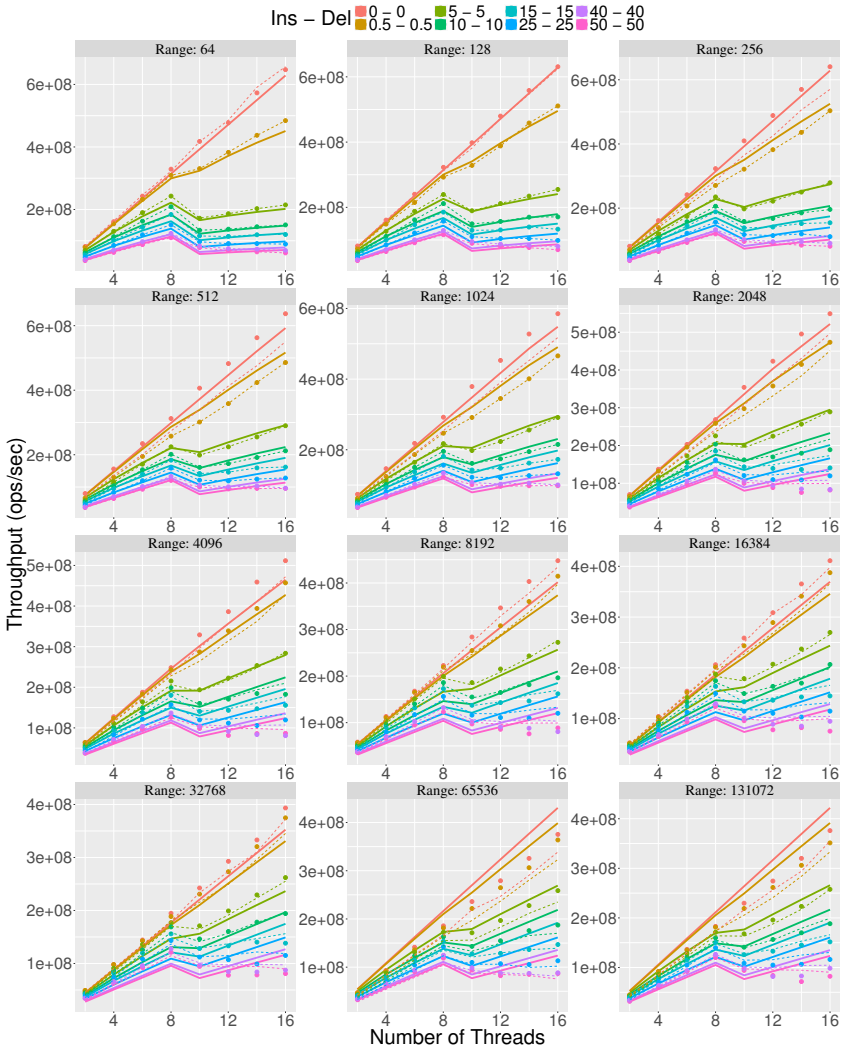


Figure 5.21: Packed nodes for Hash Table, with load factor=2

the Poisson process allowed us to consider the thread-wise and system-wise interleaving of events which are crucial for the estimation of the throughput. For

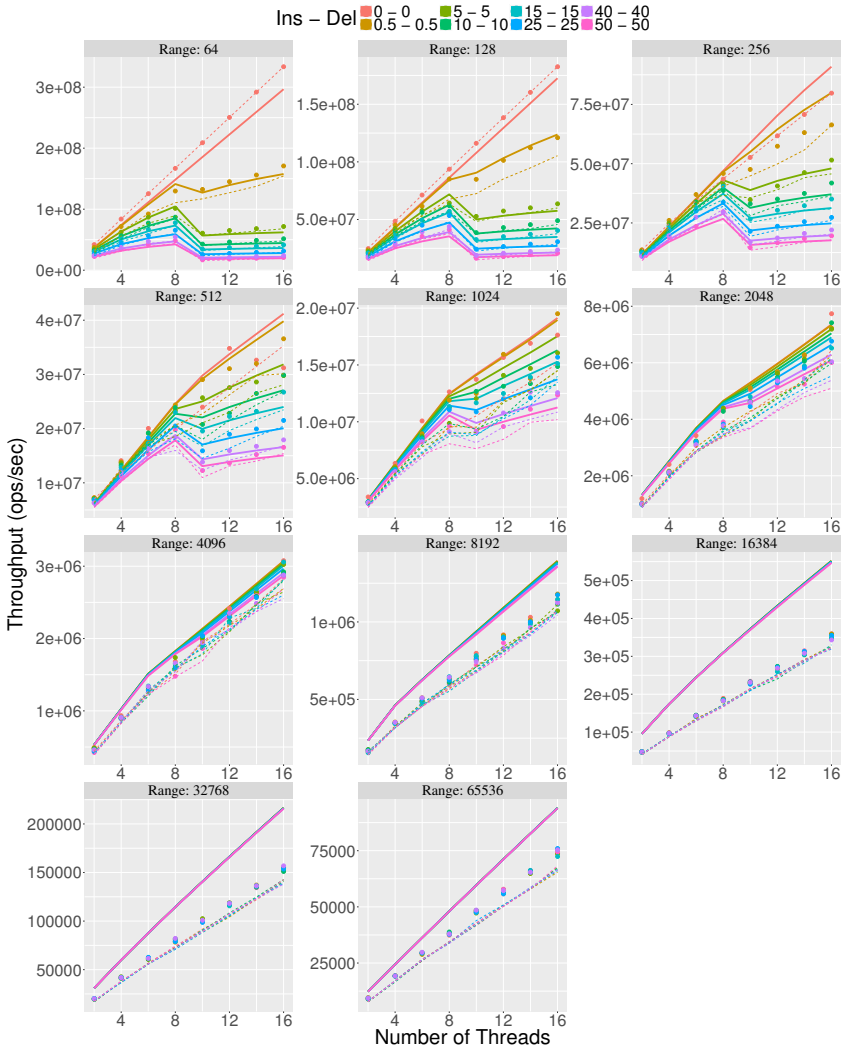


Figure 5.22: Packed nodes for Linked List

the validation, we have used several fundemental lock-free search data structures.

As the future work, it would be of interest to study to which extent the application workload can be distorted while giving satisfactory results. Putting aside the non-memoryless access patterns, the non-stationary workloads such as bursty access patterns, could be covered by splitting the time interval into alternating phases and assuming a stationary behaviour for each phase. Furthermore, we foresee that the framework can capture the performance of lock-based search data structures and also can be exploited to predict the energy efficiency of the concurrent search data structures.

Bibliography

[1] Peter Kirschenhofer and Helmut Prodinger, "The path length of random skip lists," *Acta Informatica*, vol. 31, no. 8, pp. 775–792, 1994.

[2] William Pugh, "Skip lists: A probabilistic alternative to balanced trees," *Communications of the ACM*, vol. 33, no. 6, pp. 668–676, 1990.

[3] Luc Devroye, "A note on the height of binary search trees," *Journal of the ACM (JACM)*, vol. 33, no. 3, pp. 489–498, 1986.

[4] Raimund Seidel and Cecilia R. Aragon, "Randomized search trees," *Algorithmica*, vol. 16, no. 4/5, pp. 464–497, 1996.

[5] Hosam M. Mahmoud and Ralph Neininger, "Distribution of distances in random binary search trees," *The Annals of Applied Probability*, vol. 13, no. 1, pp. 253–276, 2003.

[6] Yutao Zhong, Steven G. Dropsho, Xipeng Shen, Ahren Studer, and Chen Ding, "Miss rate prediction across program inputs and cache configurations," *IEEE Transactions on Computers (TC)*, vol. 56, no. 3, pp. 328–343, 2007.

[7] Philippe Flajolet, Danièle Gardy, and Loÿs Thimonier, "Birthday paradox, coupon collectors, caching algorithms and self-organizing search," *Discrete Applied Mathematics*, vol. 39, no. 3, pp. 207–229, 1992.

[8] James D. Fix, "The set-associative cache performance of search trees," in *Proceedings of the ACM-SIAM Symposium On Discrete Algorithms (SODA)*. 2003, pp. 565–572, ACM/SIAM.

[9] Mikhail Fomitchev and Eric Ruppert, "Lock-free linked lists and skip lists," in *Proceedings of the ACM Symposium on Principles of Distributed Computing (PoDC)*. 2004, pp. 50–59, ACM.

[10] Bapi Chatterjee, Nhan Nguyen Dang, and Philippas Tsigas, "Efficient lock-free binary search trees," in *Proceedings of the ACM Symposium on Principles of Distributed Computing (PoDC)*. 2014, pp. 322–331, ACM.

[11] Vincent Gramoli, "More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms," in *Principles and Practice of Parallel Programming (PPoPP)*. 2015, pp. 1–10, ACM.

[12] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis, "Asynchronized concurrency: The secret to scaling concurrent search data structures," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2015, pp. 631–644, ACM.

[13] Tudor David and Rachid Guerraoui, "Concurrent search data structures can be blocking and practically wait-free," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2016, pp. 337–348, ACM.

[14] Andrew Barbour and Timothy Carlisle Brown, "Stein's method and point process approximation," *Stochastic Processes and their Applications*, vol. 43, no. 1, pp. 9 – 31, 1992.

[15] Louis Chen and Adrian RÃűllin, "Approximating dependent rare events," *Bernoulli*, vol. 19, no. 4, pp. 1243–1267, 2013.

[16] Richard Arratia, Larry Goldstein, and Louis Gordon, "Poisson approximation and the chen-stein method," *Statistical Science*, vol. 5, no. 4, pp. 403–424, 1990.

[17] Timothy Carlisle Brown, Graham Weinberg, and Aihua Xia, "Removing logarithms from poisson process error bounds," *Stochastic Processes and their Applications*, vol. 87, no. 1, pp. 149 – 165, 2000.

[18] Christine Fricker, Philippe Robert, and James Roberts, "A versatile and accurate approximation for LRU cache performance," *CoRR*, vol. abs/1202.3974, 2012.

[19] John D. C. Little, "A proof for the queuing formula: L= λ w," *Operations research*, vol. 9, no. 3, pp. 383–387, 1961.

[20] Timothy L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the International Symposium on Distributed Computing (DISC)*. 2001, vol. 2180 of *Lecture Notes in Computer Science*, pp. 300–314, Springer.

[21] Håkan Sundell and Philippas Tsigas, "Fast and lock-free concurrent priority queues for multi-thread systems," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 65, no. 5, pp. 609–627, 2005.

[22] Aravind Natarajan and Neeraj Mittal, "Fast concurrent lock-free binary search trees," in *Principles and Practice of Parallel Programming (PPoPP)*. 2014, pp. 317–328, ACM.

[23] Gabriele Paoloni, "How to benchmark code execution times on Intel® ia-32 and ia-64 instruction set architectures," Tech. Rep. 324264-001, Intel, 2010.

[24] Vlastimil Babka and Petr Tuma, "Investigating cache parameters of x86 family processors," in *SPEC Benchmark Workshop*. 2009, vol. 5419 of *Lecture Notes in Computer Science*, pp. 77–96, Springer.

Part III

CONCLUSION

6

Conclusion and Future Work

In this thesis, we have presented analytical methodologies to estimate the throughput and energy efficiency of lock-free data structures. Our models cover lock-free designs of various abstract data types that are exploited in a wide range of scenarios including several contention levels, access patterns, number of threads, hardware configurations, data structure sizes.

We validate our models in a broad spectrum of data structures implementations such as queues, stacks, priority queues, hash tables, skip lists, deques, hash tables, binary trees, linked lists and obtain performance estimates that are close to what we observe in practice. Besides, we make use of our analyses to: (i) design a new back-off strategy; (ii) optimize memory management mechanism; (iii) resolve the impact of different memory alignment strategies.

For the future work, we envision two extensions to our models. Firstly, we

can adapt our frameworks to analyze the performance of lock-based concurrent algorithms. For example, we can estimate the performance of a Test-And-Set lock or a lock-based search data structure by using our existing frameworks. These examples show some promise that our frameworks can be extended to cover algorithms with various locking mechanisms.

Lock-based approaches are easier to understand and implement compared to the lock-free ones. Therefore, there is widespread interest from programmers to the lock-based approaches when it comes to implementing concurrent programs. This extension would appeal to a broader audience than their lock-free variants and provide an understanding of the performance of lock-based synchronization mechanisms. Also, it would be interesting to compare the energy efficiency and throughput of lock-free and lock-based approaches. It might be possible to determine the characteristics of the configurations where one approach is better than the other. This can lead to new designs for concurrent data structures and also can help to determine the data structure that suits best to the application at hand.

Secondly, we can extend our frameworks to estimate the energy efficiency of search data structures. With micro-benchmarking, we can extract the energy consumption cost for each type of event that is defined in our model. Then, we can couple these values with the occurrence rate of events that are provided by our throughput framework. This approach would lead to the energy consumption of the search data structures.

On the other hand, we can focus on the applications of our analyses. We have predicted but have not yet attempted to improve the energy efficiency of lock-free data structures. Back-off is a well-known strategy that is often used to improve the performance. We believe, an emphasis on energy efficiency can also be put in the back-off mechanisms. A subset of processors can apply *DVFS* technique to back-off by reducing their clock frequency. This approach would improve not only the performance but also the energy efficiency.

Another way to improve energy efficiency is to avoid wasteful use of resources. In this thesis, we have considered programs that are parallelized based on producer/consumer pattern. For this case, the number of consumer processes

can be tuned according to the production rate of producer processes since an imbalance of the rates might lead to waste of resources. Our framework can be used to predict consumption and production rates (throughput) and to set the number of producer and consumer processes to minimize the waste of resources.

We have observed that inherent sequential bottlenecks limit the scalability of some abstract data types (*e.g.* stack, queue). To overcome this limitation, techniques like elimination, combining, semantic relaxation can be used. However, these techniques are only useful when the contention on the data structure is high enough and turns out to be a burden otherwise. Think of a concurrent stack that employs elimination: a push operation declares its operation in an entry of an elimination array and waits for some time for a matching pop operation that searches the array. If a matching operation is not found, the operation falls back to the original path and takes place on the concurrent stack. Here, the size of the elimination array is crucial for performance since small sizes might restrict the scalability under very high contention, and large sizes could be harmful by increasing the search time for the pop operation. A similar logic applies to the semantic relaxation concerning the accuracy metric. A stack can be relaxed to allow for concurrent accesses from many processes. Unless there is enough concurrency, relaxation might decrease the accuracy but would be ineffective to increase the performance. Our results can be extended to cover these designs and can be used to eliminate the mentioned losses in the performance and accuracy metrics.